Tutorial 7

CS2106: Introduction to Operating Systems

Mem A | B | ....

Contiguous Memory Management

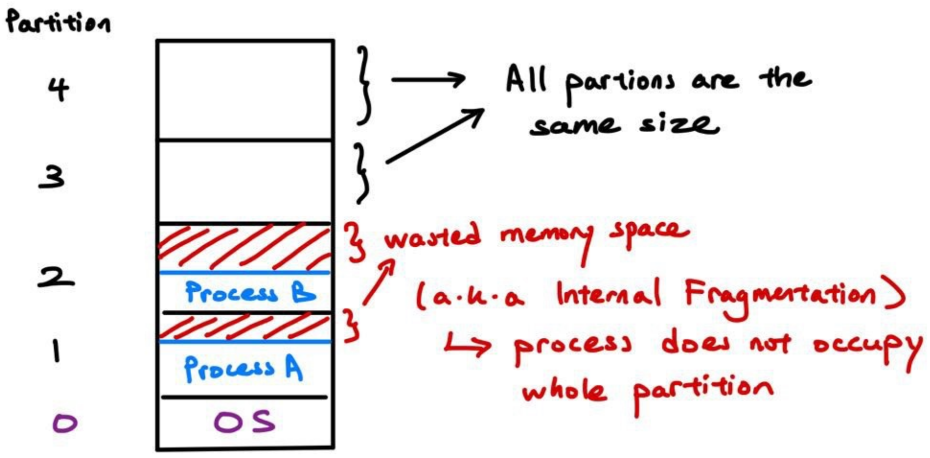
- In the lecture, we make the following two assumptions
  1. Each process occupies a **contiguous memory region**
  2. The physical memory is large enough to contain one or more processes with complete memory space.
- Process must be in memory during execution
  - Store Memory concept
  - Load-Store Memory execution model

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 → Hard Disk / SSD

Partition Allocation Schemes

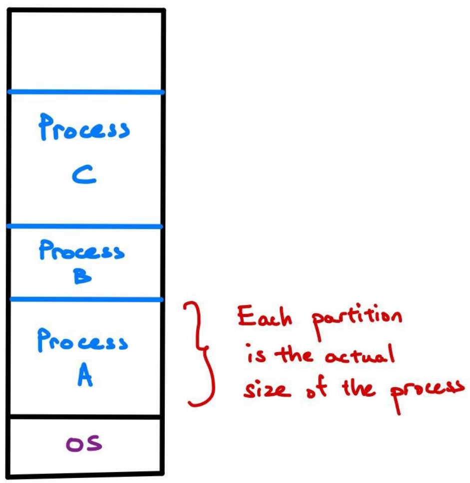
Fixed Partitioning



Pros:
1. Easy to manage
2. Easy to allocate

Cons:
1. Partition size needs to be large enough to contain the largest process.
2. Smaller processes will waste memory space

Dynamic Partitioning



Pros:
1. No internal fragmentation: All processes get the exact space it requires.

Cons:
1. Need to maintain more information in OS
2. Takes more time to locate appropriate region
3. Prone to external fragmentation

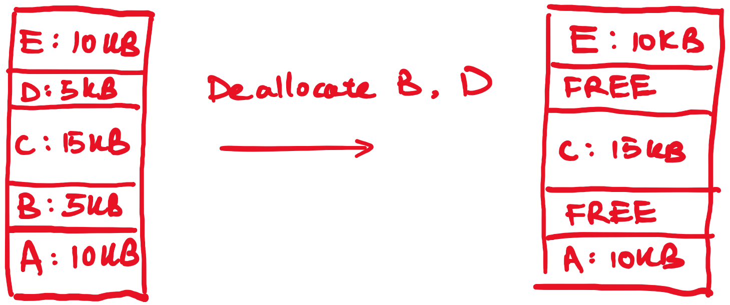
Internal and External Fragmentation

max no. of allocations

- Fixed Partitioning cannot have external fragmentation.
  - Memory is divided into fixed-size partitions. → fixed no. of partitions
  - Each partition is allocated to a single process.
- Dynamic partitioning cannot have internal fragmentation.
  - The exact amount of memory requested by the process is allocated

Note: Internal Fragmentation: Process is allocated more memory than it needs

External Fragmentation:



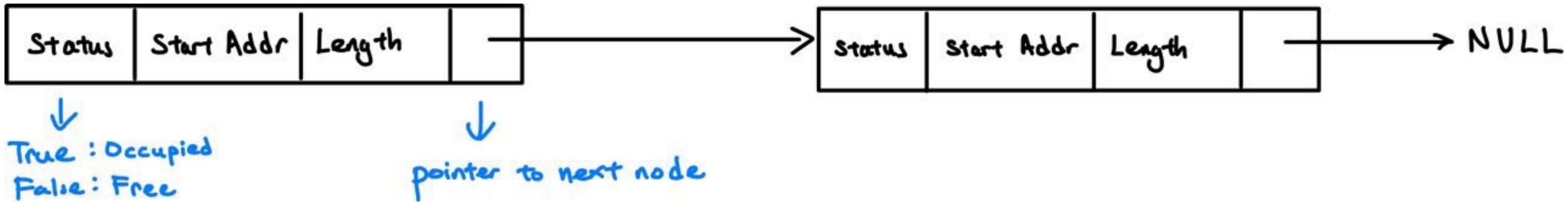
Allocate F: 10KB

More on Dynamic Partitioning

- Free memory space is also known as a “hole”.
- More holes are created each time a process is created, terminated or swapped.
- OS maintains a linked list of **partition information**
  - Perform splitting and merging when necessary
- When an occupied partition is freed
  - Merge with adjacent hole if possible
  - Compaction can also be used to consolidate holes, but it is time consuming.

Dynamic Partitioning: Partition Info

- Can be maintained as a linked list or bitmap.
  - Linked List

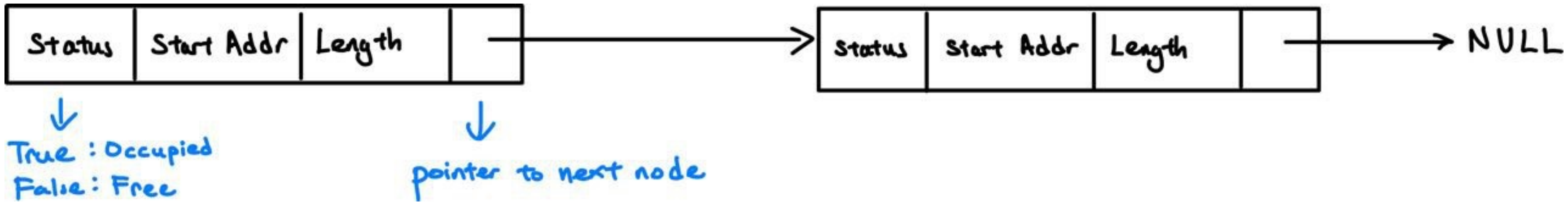


  - Bitmap
    - Tutorial 7 Question 1

Question 1

Question 1: Alternative to Linked List

In the lecture, we used linked list to store partition information under the dynamic allocation scheme.



Question 1: Alternative to Linked List

One common alternative is to use bitmap (array of bits) instead.

- Basic idea: A single bit represents the smallest allocatable memory space, 0 = free, 1 = occupied.
- Use a collection of bits to represent the allocation status of the whole memory space.

Question 1: Alternative to Linked List

As a tiny example, suppose the memory size is 16KB and the smallest allocatable unit is 1KB. We need 16 bits (2 bytes) to keep track of the allocation status:

Allocation Status

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Corresponding physical memory layout at this point.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- At this point the whole memory is a single free partition.
- The boxes are drawn to illustrate the allocation unit clearly.

Question 1: Alternative to Linked List

After placing process A (6KB), the bitmap and the corresponding physical memory layout become:

Allocation Status

1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0

Corresponding physical memory layout

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A															

Question 1: Alternative to Linked List

Give brief pseudo code to:

a) Allocate X KB using the bitmap using **first-fit.**

b) Deallocate (free) X KB with start location Y.

c) Merge adjacent free space.

Take the first hole that is large enough

Free Y, Y+1 ..., Y+X-1

Note: For ease of discussion, the pseudo codes are presented as if array indexing is provided for bitmap.

Question 1(a)

Allocate X KB using the bitmap using **first-fit**.

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0

1	Start ← 0
2	Start ← Location of the first '0' in the bitmap after Start
3	If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
4	Else Start ← The first '1' in the bitmap after Start, repeat step 2.
5	Stop when bitmap is exhausted.

Question 1(a)

Example:
Allocate 5 KB

Allocate X KB using the bitmap using **first-fit**.

Start

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0

1	Start ← 0
2	Start ← Location of the first '0' in the bitmap after Start
3	If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
4	Else Start ← The first '1' in the bitmap after Start, repeat step 2.
5	Stop when bitmap is exhausted.

Start = 0

Question 1(a)

Example:
Allocate 5 KB

Allocate X KB using the bitmap using **first-fit**.

Start

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0

1	Start ← 0
2	Start ← Location of the first '0' in the bitmap after Start
3	If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
4	Else Start ← The first '1' in the bitmap after Start, repeat step 2.
5	Stop when bitmap is exhausted.

Start = 6

Question 1(a)

Example:
Allocate 5 KB

Allocate X KB using the bitmap using **first-fit**.

Start

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0

1	Start ← 0
2	Start ← Location of the first '0' in the bitmap after Start
3	If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
4	Else Start ← The first '1' in the bitmap after Start, repeat step 2.
5	Stop when bitmap is exhausted.

Start = 6

There isn't 5 consecutive zeroes

Question 1(a)

Example:
Allocate 5 KB

Allocate X KB using the bitmap using **first-fit**.

Start

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0

1	Start ← 0
2	Start ← Location of the first '0' in the bitmap after Start
3	If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
4	Else Start ← The first '1' in the bitmap after Start, repeat step 2.
5	Stop when bitmap is exhausted.

Start = 10

Question 1(a)

Example:
Allocate 5 KB

Allocate X KB using the bitmap using **first-fit**.

Start

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1

1	Start ← 0
2	Start ← Location of the first '0' in the bitmap after Start
3	If there are X consecutive zeroes, mark [Start…Start+X-1] as 1, success!
4	Else Start ← The first '1' in the bitmap after Start, repeat step 2.
5	Stop when bitmap is exhausted.

Start = 11

Success! There are 5 consecutive zeroes

Question 1(b)

Deallocate (free) X KB with start location Y.

BEFORE

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1

Straightforward, mark [Y… Y+X-1] as 0. Done!

Example:
Deallocate 6KB with start location 0

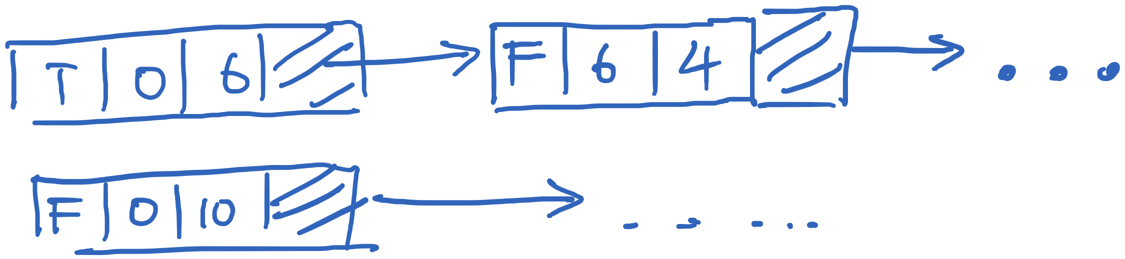
AFTER

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Allocation Status	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

Question 1(c)

compare to linked-list representation of partition-info

Example:
Deallocate 6KB with start location 0



Merge adjacent free space.

- No need to do anything ☺️.
- This is the benefit of using bitmap as adjacent free spaces are "merged" automatically.

BEFORE

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	1	1	1	1	1	1	0	0	0	0	1	1	1	1	1	1

AFTER

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Allocation Status	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1

Question 2

Question 2: Memory Allocation

Consider six memory partitions of size 200 KB, 400 KB, 600 KB, 500 KB, 300 KB and 250 KB.

200KB	400KB	600KB	500KB	300KB	250KB

These partitions need to be allocated to four processes:

P_1=357KB, P_2=210KB, P_3=468KB, P_4=491KB in that order.

Perform the allocation of processes using:

a) First Fit Algorithm
b) Best Fit Algorithm
c) Worst Fit Algorithm

Note: The main memory in this question has been divided into fixed size partitions

Memory Allocation Algorithms

1. First Fit: Take the first hole that is large enough
2. Best Fit: Take the smallest hole that is large enough
3. Worst Fit: Take the largest hole

Question 2: Memory Allocation

P_1=357KB, P_2=210KB, P_3=468KB, P_4=491KB

1. First Fit: Take the first hole that is large enough
2. Best Fit: Take the smallest hole that is large enough
3. Worst Fit: Take the largest hole

- First Fit

200KB	400KB	600KB	500KB	300KB	250KB

- Best Fit

200KB	400KB	600KB	500KB	300KB	250KB

- Worst Fit

200KB	400KB	600KB	500KB	300KB	250KB

Question 2: Memory Allocation

- First Fit

200KB	400KB	600KB	500KB	300KB	250KB
	P_1=357KB	P_2=210KB	P_3=468KB		

- Best Fit

200KB	400KB	600KB	500KB	300KB	250KB
	P_1=357KB	P_4=491KB	P_3=468KB		P_2=210KB

- Worst Fit

200KB	400KB	600KB	500KB	300KB	250KB
		P_1=357KB	P_2=210KB		

Question 2

Which algorithm makes the most efficient use of memory in this particular case?

- Best Fit Algorithm turns out to be the best in terms of memory efficiency in this case

Which algorithm has the best average runtime?

- Regarding the runtime, Best Fit and Worst Fit must go through the entire list, taking O(N) to find the best (worst) candidate.
- First Fit has the best runtime as the search stops as soon as the first free hole that accommodates the request is available.

Question 3

Question 3: Modified First Fit

- Suppose we use the following allocation algorithm: for the first allocation, traverse the list of free partitions from the beginning of the list until the first partition which can accommodate the request.
- The following allocations, however, do not start from the beginning, but instead **start from the partition where the previous allocation was performed**.
- Compare this algorithm with **First Fit** in terms of **runtime** and **efficiency of memory use**.

Question 3: Modified First Fit

- Suppose we have a contiguous memory region which uses **dynamic partitioning scheme**.
- Currently, the memory region looks like this after multiple process allocations and deallocations.

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Total: 8MB = 8192KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let’s allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let’s allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

First Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	1000KB (Occ)	1500KB (Free)

Allocate P_2=1000KB

Question 3

- *First Fit* always starts the search from the beginning of the free list. Over time, the holes close to the beginning will become too small, so the algorithm will have to look further down the list, increasing the search time.
- The described algorithm is known as *Next Fit* and avoids the above problem by changing the starting point of the search. This in turn leads to a more uniform distribution of hole sizes across the free list and will therefore lead to **faster allocation**.

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occupied)	800KB (Free)	800KB (Occupied)	1492KB (Free)	1000KB (Occupied)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_1=1200KB

Question 3: Modified First Fit

- Let’s allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

start from the partition where the previous allocation was performed

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	2500KB (Free)

Allocate P_2=1000KB

Question 3: Modified First Fit

- Let's allocate the following processes: P_1=1200KB, P_2=1000KB

Next Fit

Total: 8MB = 8192KB

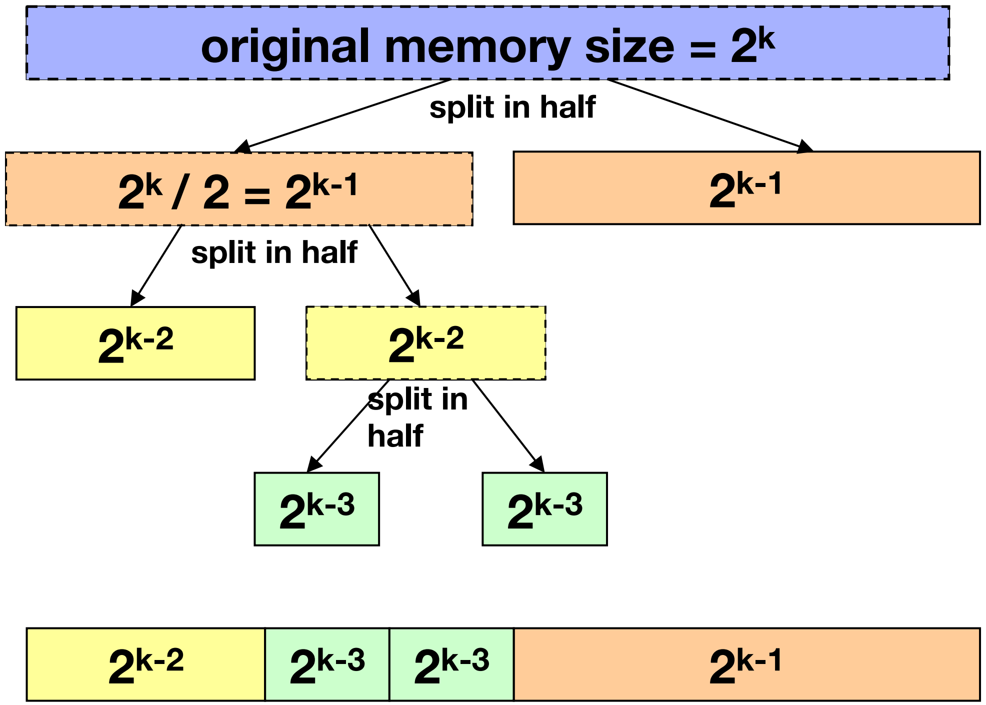
1600KB (Occ)	800KB (Free)	800KB (Occ)	1200KB (Occ)	292KB (Free)	1000KB (Occ)	1000KB (Occ)	1500KB (Free)

Allocate P_2=1000KB

Question 4

Buddy System

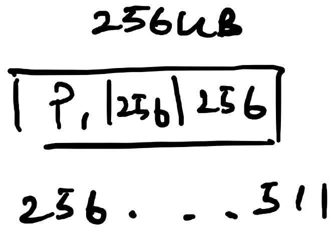
memory is in powers of 2.

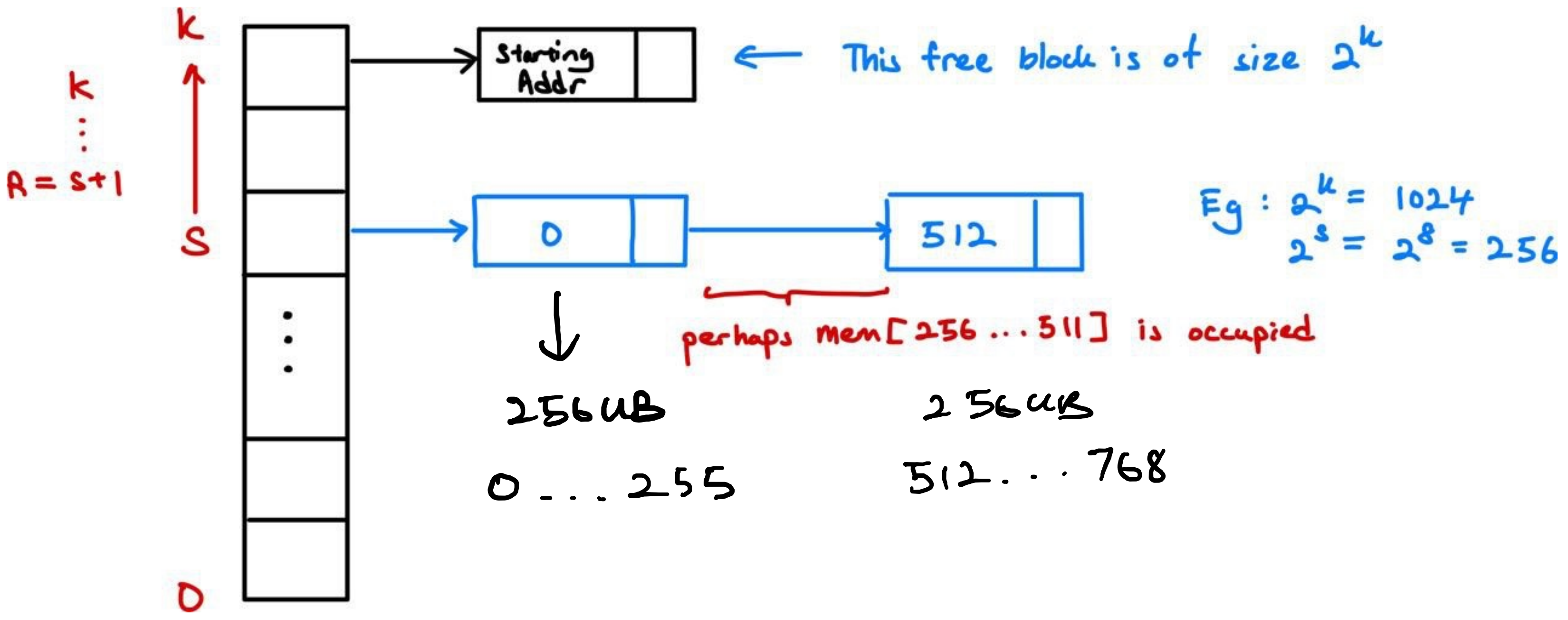


- Idea
  - Free block is repeatedly split into half to meet the request size.
  - Two halves form as buddy blocks.
  - When buddy blocks are free, they are merged to form larger blocks
- Benefits
  - Efficient partition splitting, locating of free holes, deallocation and merging.

Buddy System: Implementation

- Array of size k where 2^k is the largest allocatable block size.
  - Each array element **`A[S]`** is a linked list which keeps track of free blocks of the size $\mathbf{2^S}$.
  - Each free block is indicated by just the starting address.





Buddy System

Allocation Algorithm

1. Find smallest S such that $2^S \geq N$
2. Access A[S] to check if a free block exists
   a) If free block exists
      - Remove free block from free block list
      - Allocate the block

   b) Else
      - Find a bigger free block B by finding the smallest R from $S + 1$ to K such that A[R] has a free block B.
      - Repeatedly split B until A[S] has a free block.
      - Go to step 2.

Deallocation Algorithm

1. Check A[S] where 2^S == size of B.
2. If the buddy of B is also free (let the buddy be C)
   - Remove B and C from the list
   - Merge B and C to form a larger block B'.
   - Go to step 1 where B = B'
3. Else the buddy of B is not free yet.
   - Insert B into the list in A[S]

Question 4: Buddy System

Given a 1024KB memory with smallest allocatable partition of 1KB, use buddy system to handle the following memory requests.

i. Allocate: Process A (240 KB)
ii. Allocate: Process B (60 KB)
iii. Allocate: Process C (100 KB)
iv. Allocate: Process D (128 KB)
v. Free: Process A
vi. Free: Process C
vii. Free: Process B

Question 4: Buddy System

- Since the smallest allocatable partition is 1KB, lets divide the memory region into 1024 partitions and assign them indices 0 to 1023

0	1	2	3	4	…	1020	1021	1022	1023

Initially: Entire memory is free

K	
10 $2^{10} = 1024$	0 **Start Index**
9 $2^9 = 512$	
8 $2^8 = 256$	
7 $2^7 = 128$	
6 $2^6 = 64$	
…	
0 $2^0 = 1$	

The following table keeps track of blocks that are free.

Currently, there is a free block of 1024KB, which is represented by 1024 1KB partitions

- The table on the left keeps track of free blocks of size 2^0 to 2^k
- Row S represents array element A[S] which is a linked list which keeps track of free blocks of the size 2^S

Allocate **[A: 240KB]**

K	
10 $2^{10} = 1024$	0
9 $2^{9} = 512$	
8 $2^{8} = 256$	
7 $2^{7} = 128$	
6 $2^{6} = 64$	
...	
0 $2^{0} = 1$	

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[A: 240KB]**

K	
10 $2^{10} = 1024$	0
9 $2^9 = 512$	
8 $2^8 = 256$	
7 $2^7 = 128$	
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[A: 240KB]**

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	0 → 512
8 $2^8 = 256$	
7 $2^7 = 128$	
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

Split #1

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[A: 240KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	512
8 $2^{8} = 256$	0 → 256
7 $2^{7} = 128$	
6 $2^{6} = 64$	
…	
0 $2^{0} = 1$	

Split #2

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
    a. Find smallest R from S+1 to K such that A[R] has a free block
    b. For R-1 to S
        i. Split
    c. Goto 2

Allocate **[A: 240KB]**

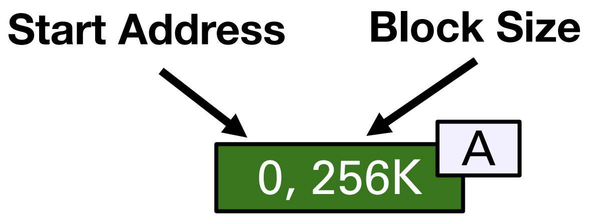
K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	512
8 $2^{8} = 256$	0 → 256
7 $2^{7} = 128$	
6 $2^{6} = 64$	
...	
0 $2^{0} = 1$	

Select Free Block

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[A: 240KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	512
8 $2^{8} = 256$	256
7 $2^{7} = 128$	
6 $2^{6} = 64$	
...	
0 $2^{0} = 1$	



Done!

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

After Allocating A

0 – 255	256 – 511	512 – 1023
A 256KB (240)	FREE 256KB	FREE 512KB

Allocate **[B: 60KB]**

0, 256K | A

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	256
7 $2^7 = 128$	
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[B: 60KB]**

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	256
7 $2^7 = 128$	
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

A

0, 256K

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
    a. Find smallest R from S+1 to K such that A[R] has a free block
    b. For R-1 to S
        i. Split
    c. Goto 2

Allocate **[B: 60KB]**

0, 256K A

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	
7 $2^7 = 128$	256 → 384
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

Split #1

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[B: 60KB]**

0, 256K A

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	
7 $2^7 = 128$	384
6 $2^6 = 64$	256 → 320
...	
0 $2^0 = 1$	

Split #2

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[B: 60KB]**

0, 256K A

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	
7 $2^7 = 128$	384
6 $2^6 = 64$	256 → 320
…	
0 $2^0 = 1$	

Select Free Block

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
    a. Find smallest R from S+1 to K such that A[R] has a free block
    b. For R-1 to S
        i. Split
    c. Goto 2

Allocate **[B: 60KB]**

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	
7 $2^7 = 128$	384
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	

0, 256K — A

256, 64K — B

Done!

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

After Allocating B

<table>
<tr><td>A
256KB
(240)</td><td colspan="3">FREE
256KB</td><td>FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td>FREE
512KB</td></tr>
</table>

Allocate **[C: 100KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	512
8 $2^{8} = 256$	
7 $2^{7} = 128$	384
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

0, 256K — A

256, 64K — B

There is one existing 128KB free partition

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
    a. Find smallest R from S+1 to K such that A[R] has a free block
    b. For R-1 to S
        i. Split
    c. Goto 2

Allocate **[C: 100KB]**

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	
7 $2^7 = 128$	384
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	

A: 0, 256K

B: 256, 64K

Select Free Block

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[C: 100KB]**

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	512
8 $2^8 = 256$	
7 $2^7 = 128$	
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	

Block	Owner
0, 256K	A
256, 64K	B
384, 128K	C

Done!

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

After Allocating C

A 256KB (240)	FREE 256KB			FREE 512KB
A 256KB (240)	B 64KB (60)	FREE 64KB	FREE 128KB	FREE 512KB
A 256KB (240)	B 64KB (60)	FREE 64KB	C 128KB (100)	FREE 512KB

256 319 384 511

Allocate **[D: 128KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	512
8 $2^{8} = 256$	
7 $2^{7} = 128$	
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

A: 0, 256K

B: 256, 64K

C: 384, 128K

Since, there is only one free partition of size 512KB, we need to split it down further

Allocate **[D: 128KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	512 → 768
7 $2^{7} = 128$	
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

A: 0, 256K

B: 256, 64K

C: 384, 128K

Split #1

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[D: 128KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	768
7 $2^{7} = 128$	512 → 640
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

A: 0, 256K

B: 256, 64K

C: 384, 128K

Split #2

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[D: 128KB]**

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	768
7 $2^{7} = 128$	512 → 640
6 $2^{6} = 64$	320
…	
0 $2^{0} = 1$	

A: 0, 256K

B: 256, 64K

C: 384, 128K

Select Free Block

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

Allocate **[D: 128KB]**

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	768
7 $2^7 = 128$	640
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	

A: 0, 256K

B: 256, 64K

C: 384, 128K

D: 512, 128K

Done!

1. Smallest S, such that $2^S >= N$
2. Does A[S] has a free block?
3. Yes: Remove from list and return
4. No
   a. Find smallest R from S+1 to K such that A[R] has a free block
   b. For R-1 to S
      i. Split
   c. Goto 2

After Allocating D

<table>
<tr><td>A
256KB
(240)</td><td colspan="3">FREE
256KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
</table>

0 255 256 319 384 511 512 629

Free A

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	768
7 $2^{7} = 128$	640
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

A: 0, 256K

B: 256, 64K

C: 384, 128K

D: 512, 128K

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B′ = B + C
   b. Goto 1 with B = B′
3. Insert B to A[S]

Free A

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 768
7 $2^{7} = 128$	640
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

B: 256, 64K

C: 384, 128K

D: 512, 128K

Does the buddy block exist?

```
0   =   0000000000
768 = 1100000000
```

8th bit

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free A

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 768
7 $2^{7} = 128$	640
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

B: 256, 64K

C: 384, 128K

D: 512, 128K

Does the buddy block exist?

```
0   =  0000000000
768 =  1100000000
```

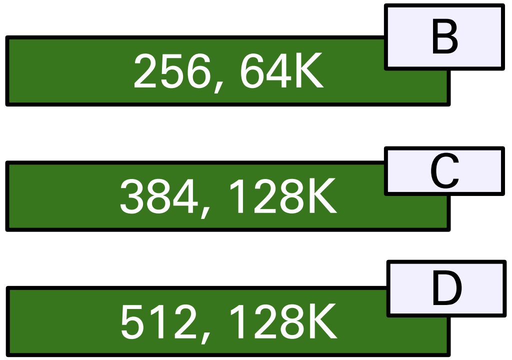
8th bit

No.

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B' = B + C
   b. Goto 1 with B = B'
3. Insert B to A[S]

Free A

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	640
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	



Insert Block into Free List

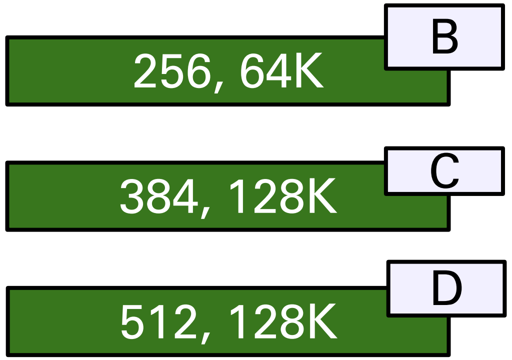
1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

After Freeing A

<table>
<tr><td>A
256KB
(240)</td><td colspan="3">FREE
256KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
<tr><td>FREE
256KB</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
</table>

Free C

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	640
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	



1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B' = B + C
   b. Goto 1 with B = B'
3. Insert B to A[S]

Free C

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 768
7 $2^{7} = 128$	384 → 640
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	

B: 256, 64K

D: 512, 128K

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free C

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	384 → 640
6 $2^6 = 64$	320
…	
0 $2^0 = 1$	

256, 64K — B

512, 128K — D

Does the buddy block exist?

```
384 = 0110000000
640 = 1010000000
```

7th bit

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
    a. Merge: B’ = B + C
    b. Goto 1 with B = B’
3. Insert B to A[S]

Free C

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	384 → 640
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	

B: 256, 64K

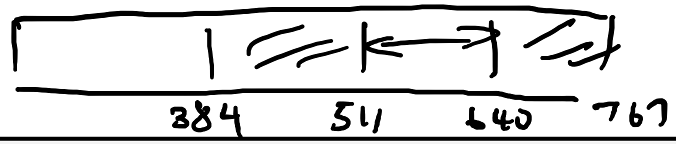
D: 512, 128K

Does the buddy block exist?

```
384 = 0110000000
640 = 1010000000
```

No.

7th bit



1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free C

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	384 → 640
6 $2^6 = 64$	320
...	
0 $2^0 = 1$	

B: 256, 64K

D: 512, 128K

Insert Block into Free List

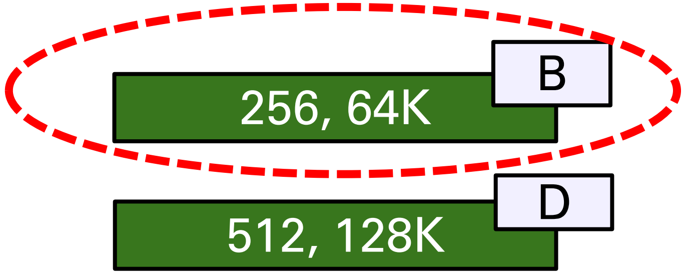
1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

After Freeing C

<table>
<tr><td>A
256KB
(240)</td><td colspan="3">FREE
256KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
<tr><td>FREE
256KB</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
<tr><td>FREE
256KB</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
</table>

Free B

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 768
7 $2^{7} = 128$	384 → 640
6 $2^{6} = 64$	320
...	
0 $2^{0} = 1$	



1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 768
7 $2^{7} = 128$	384 → 640
6 $2^{6} = 64$	256 → 320
...	
0 $2^{0} = 1$	

D

512, 128K

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	384 → 640
6 $2^6 = 64$	256 → 320
...	
0 $2^0 = 1$	

512, 128K D

Does the buddy block exist?

```
256 = 100000000
320 = 101000000
```

6th bit

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	384 → 640
6 $2^6 = 64$	256 → 320
...	
0 $2^0 = 1$	

512, 128K

D

Does the buddy block exist?

```
256 = 100000000
320 = 101000000
```

6th bit

Yes!

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B' = B + C
   b. Goto 1 with B = B'
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	256 → 384 → 640
6 $2^6 = 64$	
…	
0 $2^0 = 1$	

D

512, 128K

Merge with buddy

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 768
7 $2^7 = 128$	256 → 384 → 640
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

512, 128K

D

Does the buddy block exist?

```
256 = 100000000
384 = 110000000
```

7th bit

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 768
7 $2^{7} = 128$	256 → 384 → 640
6 $2^{6} = 64$	
...	
0 $2^{0} = 1$	

512, 128K

D

Does the buddy block exist?

```
256 = 100000000
384 = 110000000
```

7th bit

Yes!

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 256 → 768
7 $2^{7} = 128$	640
6 $2^{6} = 64$	
…	
0 $2^{0} = 1$	

512, 128K

D

Merge with buddy

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^{9} = 512$	
8 $2^{8} = 256$	0 → 256 → 768
7 $2^{7} = 128$	640
6 $2^{6} = 64$	
...	
0 $2^{0} = 1$	

512, 128K
D

Does the buddy block exist?

```
256 = 100000000
0   = 000000000
```

8th bit

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B’ = B + C
   b. Goto 1 with B = B’
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	
8 $2^8 = 256$	0 → 256 → 768
7 $2^7 = 128$	640
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

512, 128K D

Does the buddy block exist?

```
256 = 100000000
0   = 000000000
```

8th bit

Yes!

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
    a. Merge: B' = B + C
    b. Goto 1 with B = B'
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	0
8 $2^8 = 256$	768
7 $2^7 = 128$	640
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

512, 128K

D

Merge with buddy

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
    a. Merge: B' = B + C
    b. Goto 1 with B = B'
3. Insert B to A[S]

Free B

K	
10 $2^{10} = 1024$	
9 $2^9 = 512$	0
8 $2^8 = 256$	768
7 $2^7 = 128$	640
6 $2^6 = 64$	
...	
0 $2^0 = 1$	

Done!

1. Check A[S] where 2^S == size of B
2. Does the buddy C of B exist?
   a. Merge: B' = B + C
   b. Goto 1 with B = B'
3. Insert B to A[S]

After Freeing B

<table>
<tr><td>A
256KB
(240)</td><td colspan="3">FREE
256KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td colspan="3">FREE
512KB</td></tr>
<tr><td>A
256KB
(240)</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
<tr><td>FREE
256KB</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>C
128KB
(100)</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
<tr><td>FREE
256KB</td><td>B
64KB
(60)</td><td>FREE
64KB</td><td>FREE
128KB</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
<tr><td colspan="4">FREE
512KB</td><td>D
128KB
(128)</td><td>FREE
128KB</td><td>FREE
256KB</td></tr>
</table>

Question 5

Question 5: Bookkeeping Overhead

Regardless of the partitioning schemes, the kernel needs to maintain the partition information in some way (e.g. linked lists, arrays bitmaps etc). These kernel data, which is the overhead of the partitioning scheme, can consume considerable memory space.

Question 5: Bookkeeping Overhead

Given an initially free memory space of 16MB (2^{24} Bytes), briefly calculate the overhead for each of the scheme below. You should try to find a representation that reduces the overhead if possible.

For simplicity, you can assume the following size during calculation:

- Starting address, Size of partition or Pointer = 4 bytes each
- Status of partition (occupied or not) = 1 byte

Question 5(a)

1 KB = 2^{10} Bytes
1 MB = 2^{20} Bytes

Fixed-Size Partition: Each partition is 4KB size (2^{12}). What is minimum and maximum overhead?

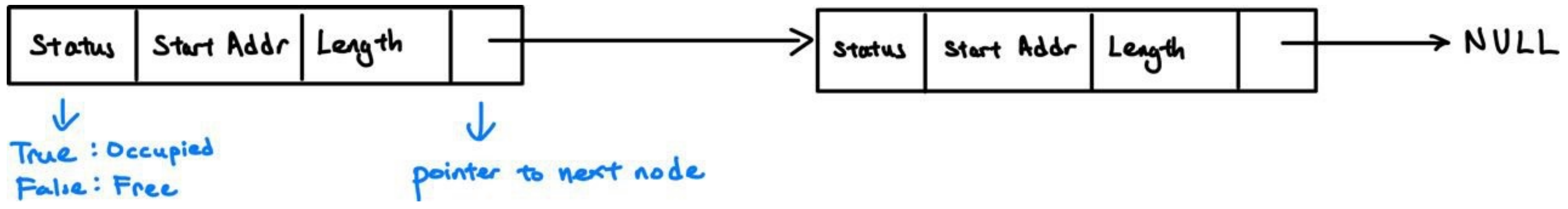
- There are (16MB/4KB = $\frac{2^{24}}{2^{12}} = 2^{12}$) partitions in this scheme.
- The simplest representation is to have an array of 2^{12} entries, each entry represent the status of a partition (occupied or free).
- Total overhead = 2^{12} entries * 1byte each = 4096 bytes
- As the partition number is fixed, maximum overhead = minimum overhead

Question 5(b)

For simplicity, you can assume the following size during calculation:
- Starting address, Size of partition or Pointer = 4 bytes each
- Status of partition (occupied or not) = 1 byte

Dynamic-Size Partition (Linked List): The smallest request size is 1KB (2^{10}), the largest request size is 4KB.

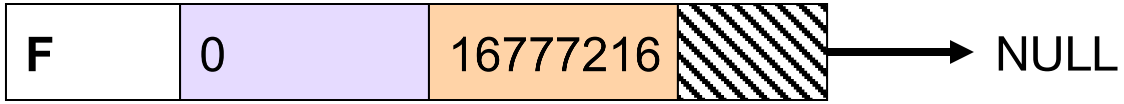
What is the minimum and maximum overhead using linked list? Allocations happen in multiples of 1 KB.



- A common linked list node structure contains
  - { Start Address, Partition Size, Status, Next Node Pointer}
- Size of one node = 3*4 + 1 = 13 bytes

Question 5(b)

- Minimum Overhead
  - When the whole partition is free, only one node is needed.
  - Overhead = 13 bytes



- Maximum Overhead
  - If every request is of the smallest size, we have the maximum number of partitions:
  - (16MB / 1KB = $\frac{2^{24}}{2^{10}}$ = 2^{14} Partitions).
  - Overhead = 2^{14} partitions * 13 bytes per node = 212922 bytes

Question 5(c)

5 KB

0 . . . 4

1 Byte = 8 Bits
1 KB = 2^{10} Bytes
1 MB = 2^{20} Bytes

Dynamic-Size Partition (Bitmap, see Q1): The smallest request size is 1KB (2^{10}), the largest request size is 4KB. What is the minimum and maximum overhead using bitmap?

- Assume that we follow Q1, such that the smallest allocatable unit is 1KB.
- The status for each allocatable unit (1KB) can be represented by each bit (1 bit) in the bitmap.
- There are $\frac{2^{24}}{2^{10}} = 2^{14}$ bits in the bitmap.

$\frac{16\text{ MB}}{1\text{ KB}} = 2^{14}$ bits

- Notice that the size of the bitmap is NOT affected by the number of partitions, hence minimum == maximum overhead.
- Overhead = 2^{14} bits / 8 bits = 2^{11} bytes = 2,048 bytes.

0	1	0	0	0	0

END OF TUTORIAL