Memory

Ravi Suppiah
Lecturer, NUS SoC

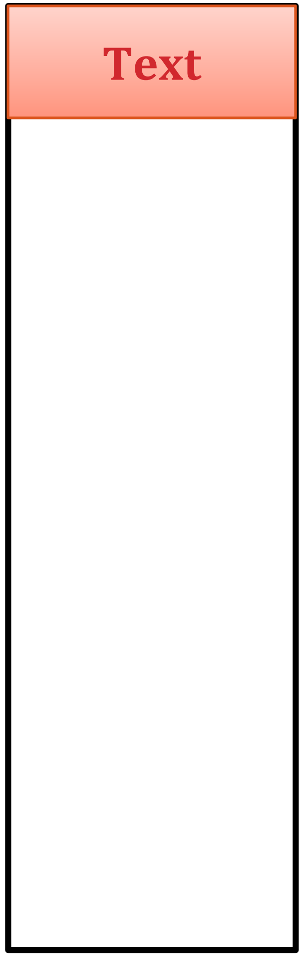
Current Presentation Content

- Different components of the process memory
  - text, data, BSS, heap, stack

- How is memory allocated to processes and the heap within a process
  - Memory allocation and deallocation
  - External and internal fragmentation
  - Memory allocation bitmap, Free list
  - Allocation policies: First fit, next fit, best fit, worst fit
  - Compaction
  - Multiple free list
  - Buddy system

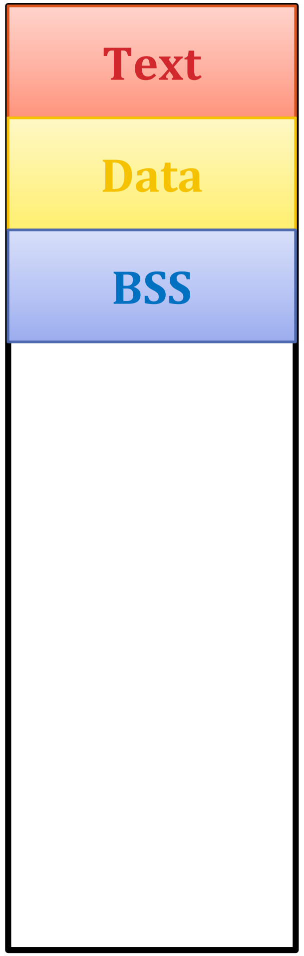
Process Memory: Code

- Executable code
  - Program binary and any other libraries it loads
- OS knows everything in advance
  - Knows amount of space needed
  - Knows the contents of the memory
- Known as the "text" segment



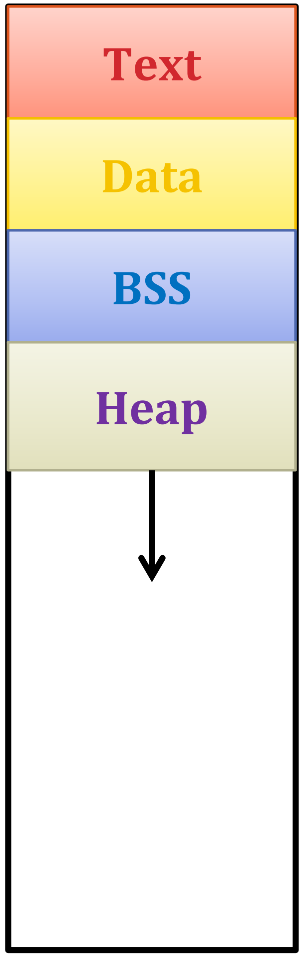
Process Memory: "Static" Data

- Variables that exist for the entire program
  - Global variables and "static" local variables
  - Amount of space required is known in advance
- Data: Initialized in the code
  - Initial value specified by the programmer
  - int x = 97;
  - Memory is initialized with this value
- BSS: not initialized in the code
  - Initial value is not specified
  - int x;
  - All memory initialized to 0
  - BSS stands for "Block Stated by Symbol"



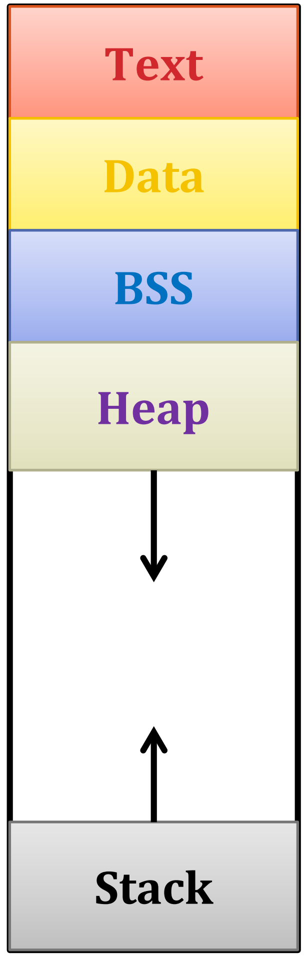
Process Memory: Dynamic Memory

- Memory allocated while program is running
  - Allocated using malloc ( ) function
  - Deallocated using free ( ) function
- OS knows nothing in advance
  - Doesn't know the amount of space
  - Doesn't know the contents
- So need to allow room to grow
  - Known as the "**heap**"



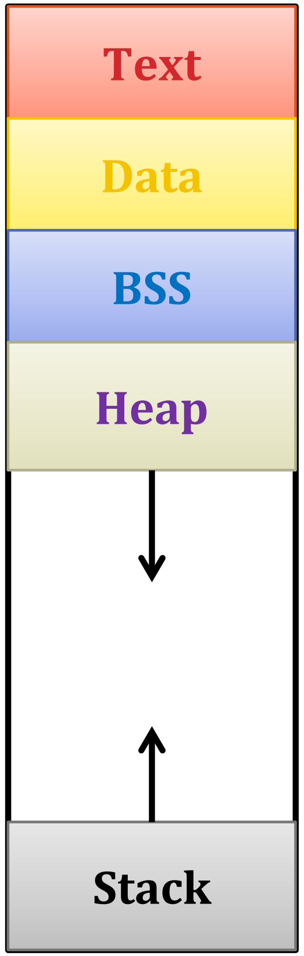
Process Memory: Temporary Variables

- Temporary memory during lifetime of a function
  - Storage for function parameters and local variables
- Need to support nested function calls
  - One function calls another
  - Store the variables of calling function
  - Know where to return when done
- So must allow room to grow
  - Known as the "**stack**"
  - Push on the stack as new function is called
  - Pop off the stack as the function ends



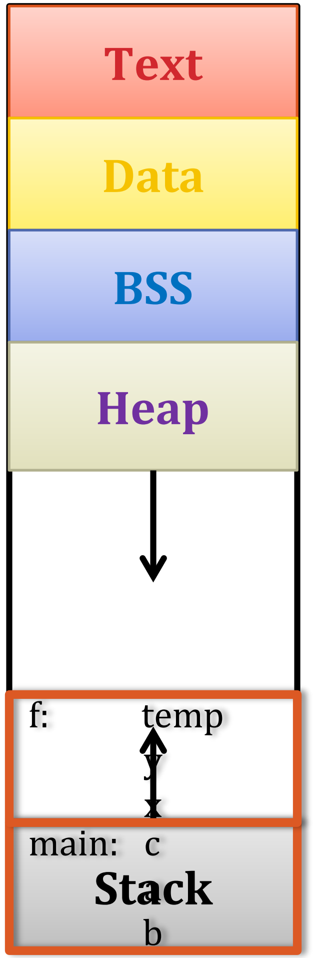
Memory Layout: Summary

- Text: Code
- Data: Initialized global and static variables
- BSS: Uninitialized global and static variables
- Heap: Dynamic Memory
- Stack: Local variables



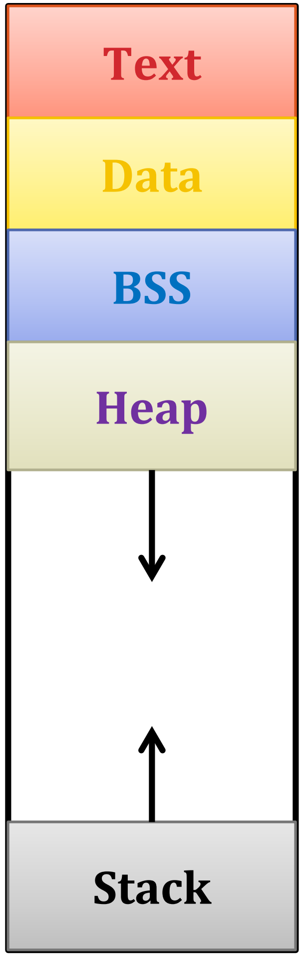
More about Stack

```
void main (void ) {
   int a=10, b=20, c;
   c = f (a, b);
}
int f (int x, int y){
  int temp;
  temp = x + y;
  return temp;
}
```



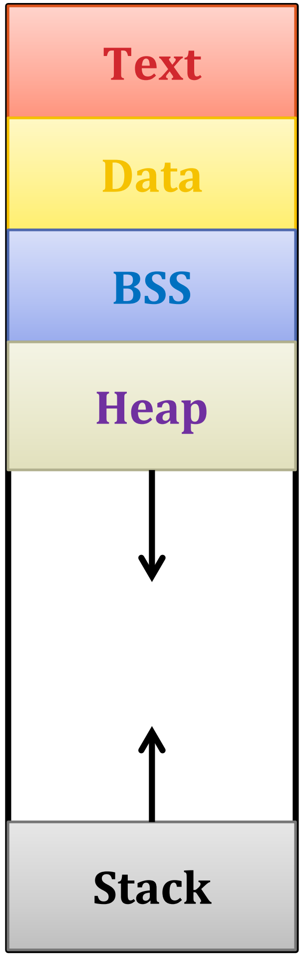
Memory Layout Example

```
int string_length = 8;
int iSize;

char * f (void)
{
        char *p;
        static int count = 0;
                iSize = string_length ;
        p = malloc(iSize);
        count ++;
        return p;
}
```



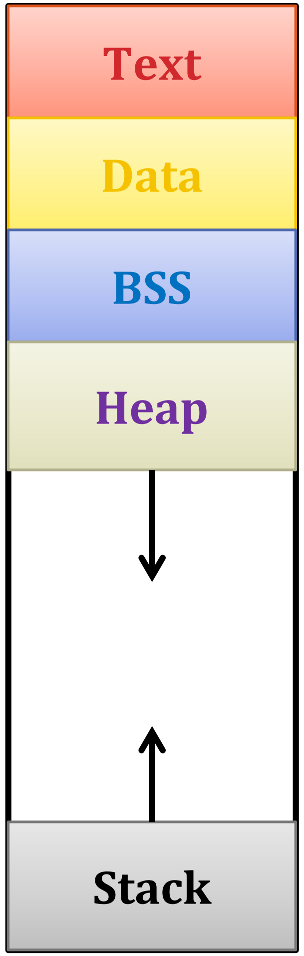
Memory Layout Example: Text

```
int string_length = 8;
int iSize;

char * f (void)
{
        char *p;
        static int count = 0;
                iSize = string_length ;
        p = malloc(iSize);
        count ++;
        return p;
}
```



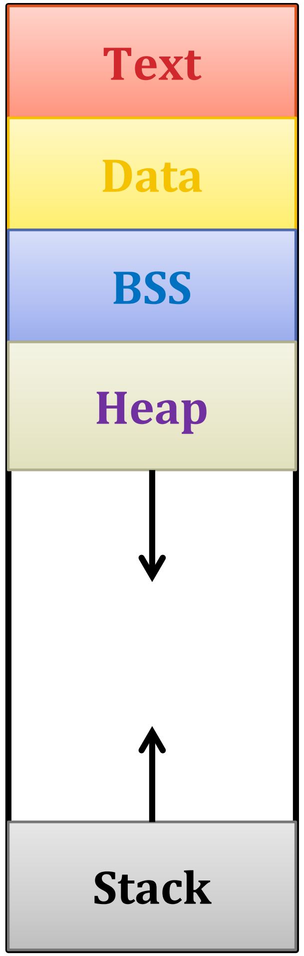
Memory Layout Example: Data

```
int string_length = 8;
int iSize;

char * f (void)
{
        char *p;
        static int count = 0;
                iSize = string_length ;
        p = malloc(iSize);
        count ++;
        return p;
}
```



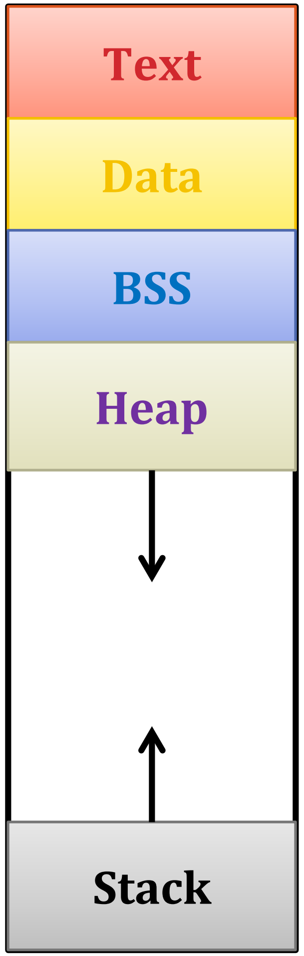
Memory Layout Example: BSS

```
int string_length = 8;
int iSize;

char * f (void)
{
        char *p;
        static int count = 0;
            iSize = string_length ;
        p = malloc(iSize);
        count ++;
        return p;
}
```



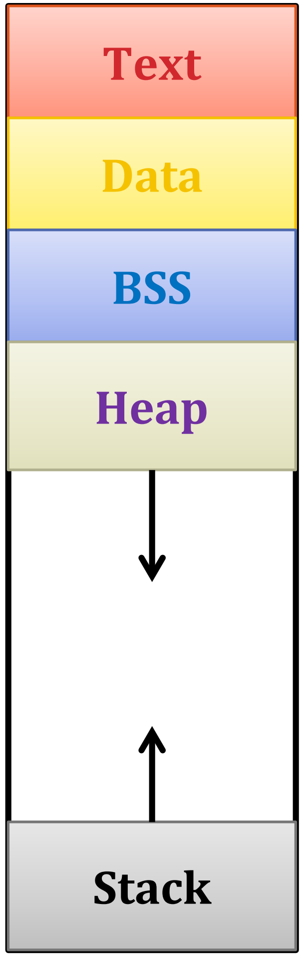
Memory Layout Example: Heap

```
int string_length = 8;
int iSize;

char * f (void)
{
        char *p;
        static int count = 0;
            iSize = string_length ;
        p = malloc(iSize);
        count ++;
        return p;
}
```



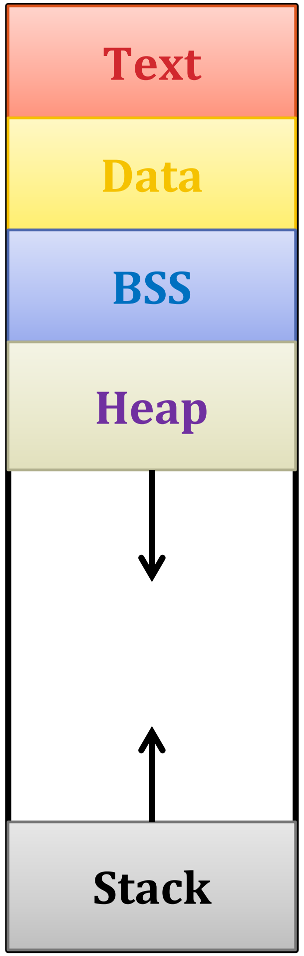
Memory Layout Example: Stack

```
int string_length = 8;
int iSize;

char * f (void)
{
        char *p;
        static int count = 0;
            iSize = string_length ;
        p = malloc(iSize);
        count++;
        return p;
}
```



Memory Allocation & Deallocation

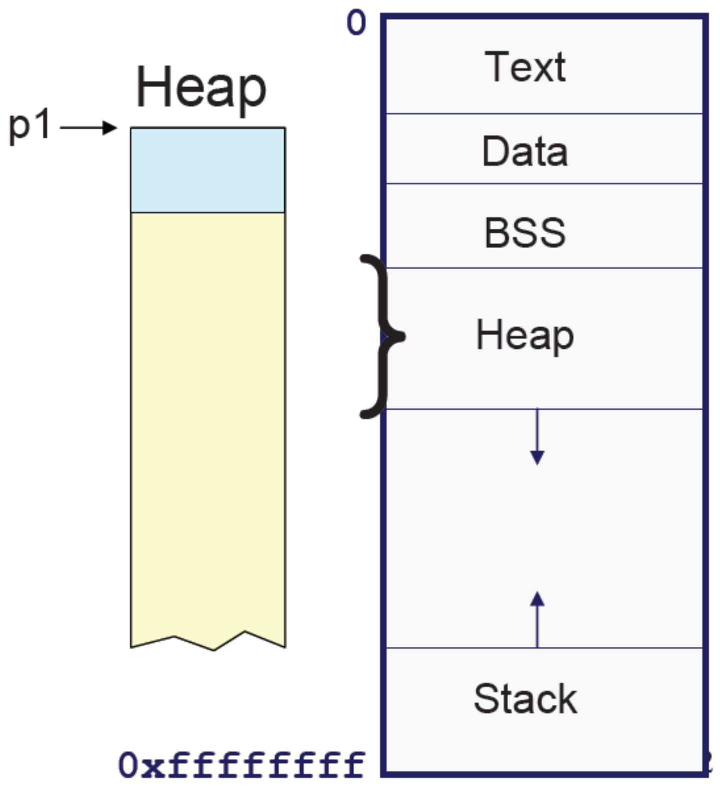
- How and when is memory allocated?
  - Global and static variable @ program startup
  - Local variables @ function call
  - Dynamic memory @ malloc
- How is memory deallocated?
  - Global and static variables @ program finish
  - Local variable @ function return
  - Dynamic memory @ free
- All memory deallocated when program ends
  - Good programming practice to free allocated memory



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

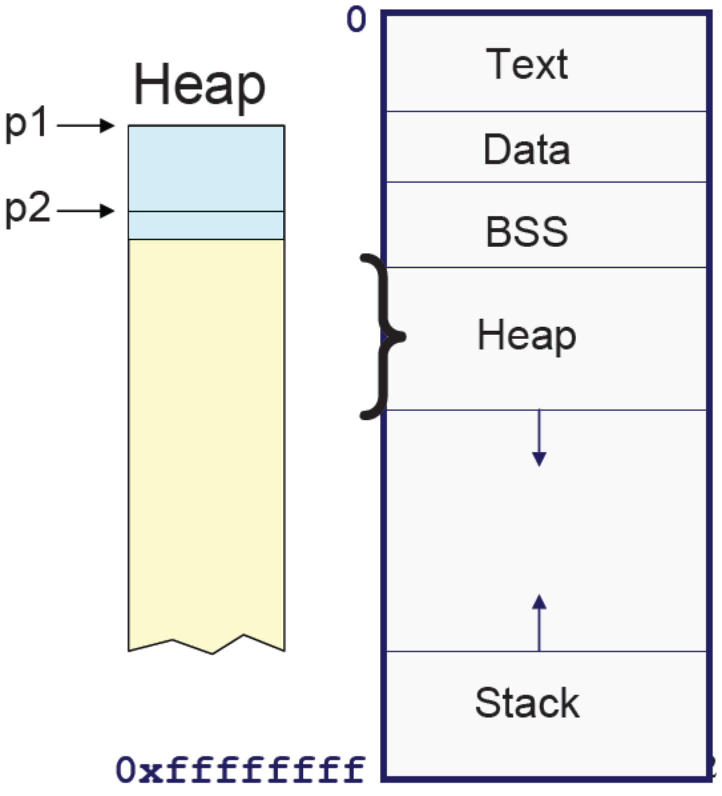
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

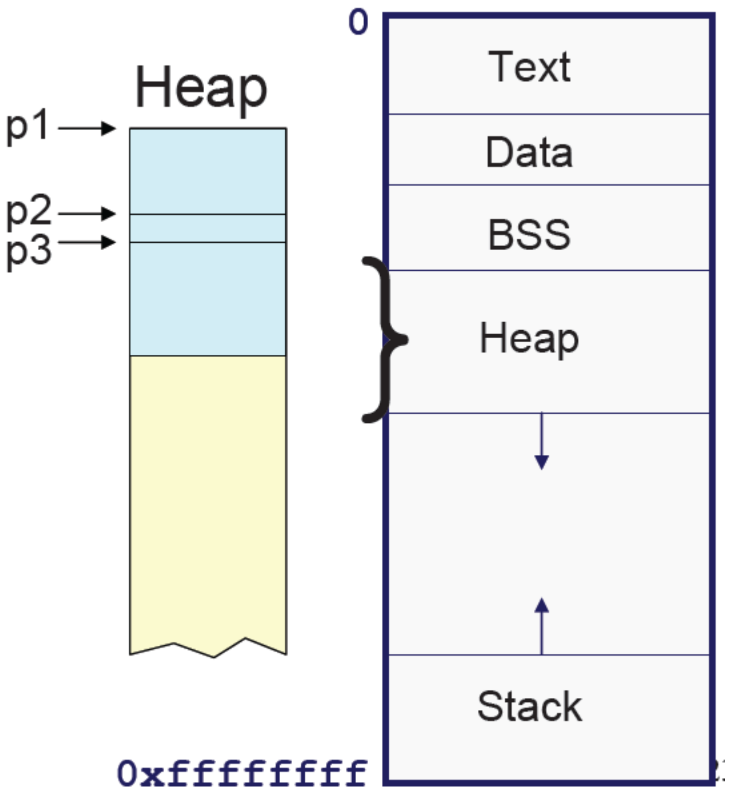
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

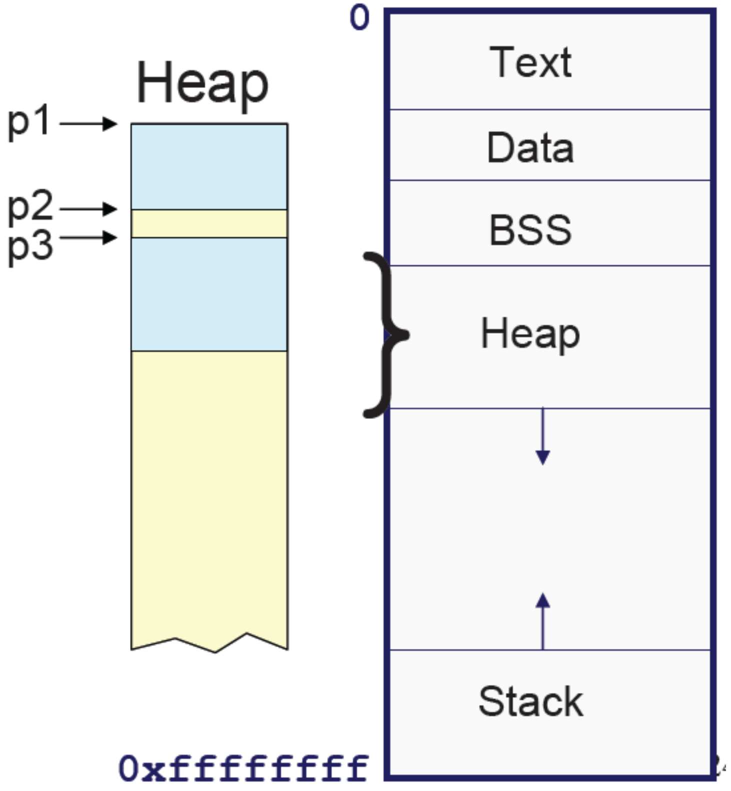
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

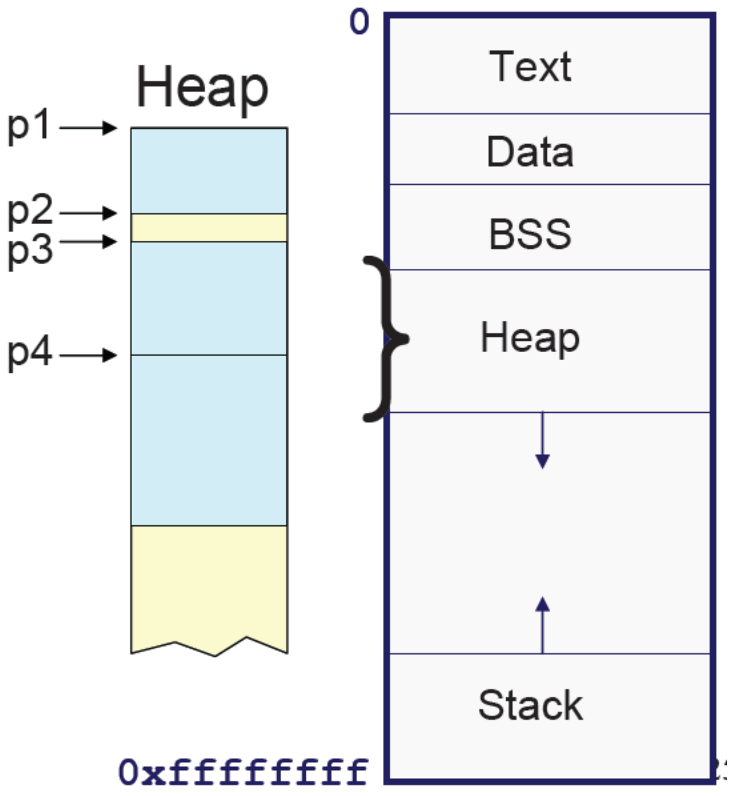
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

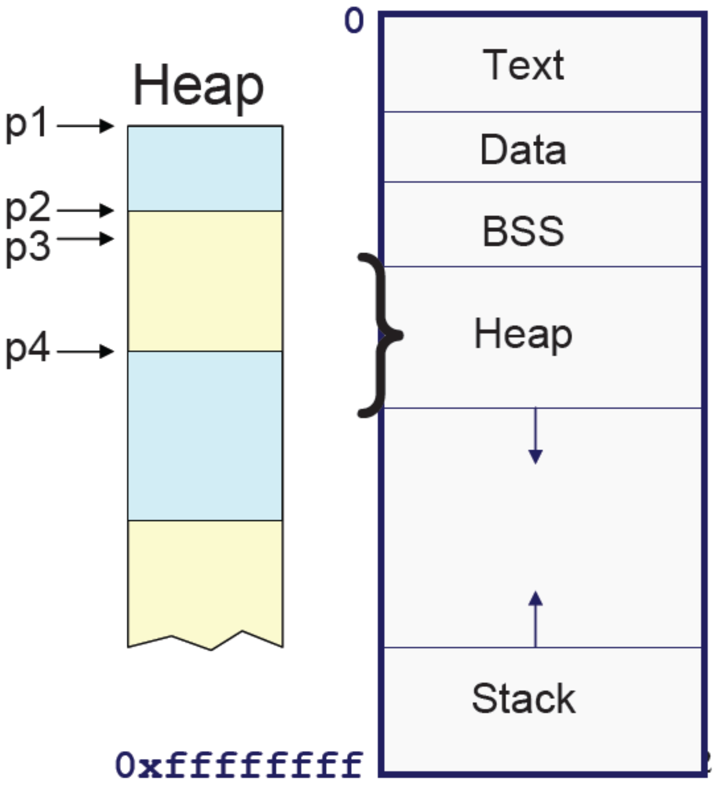
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```c
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

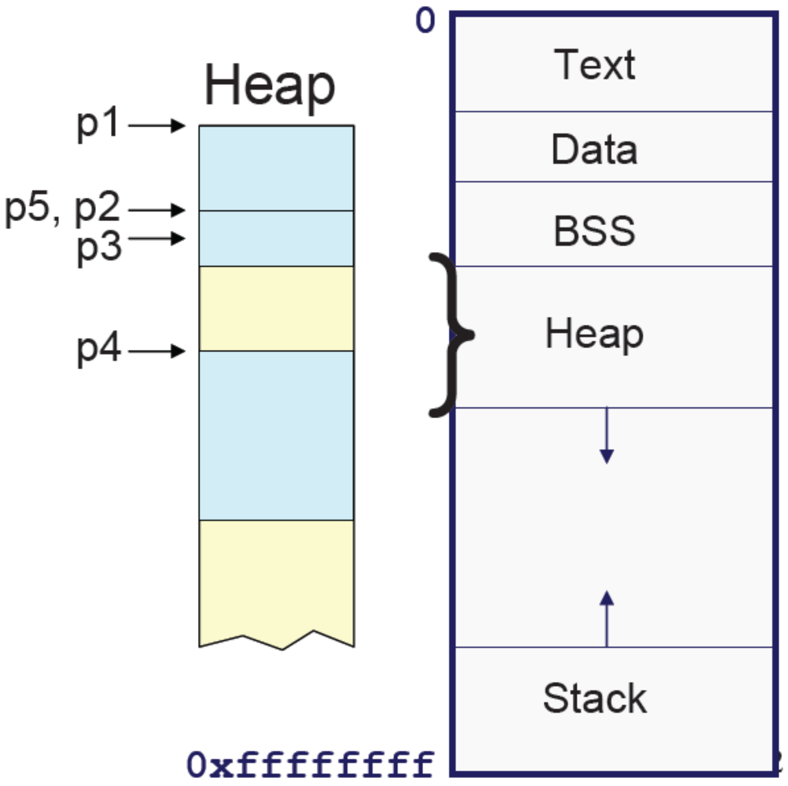
```c
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

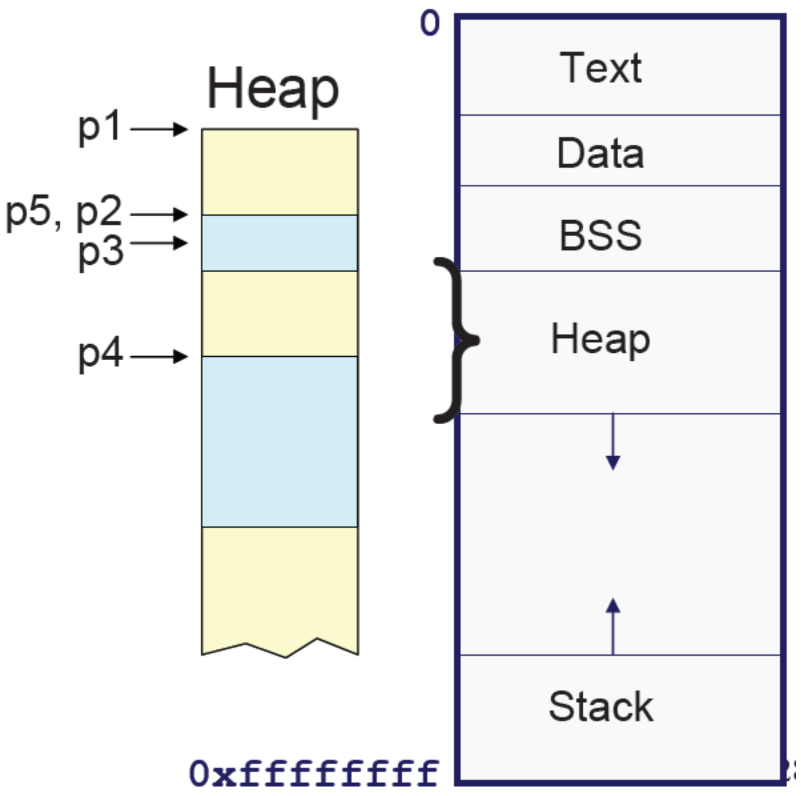
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

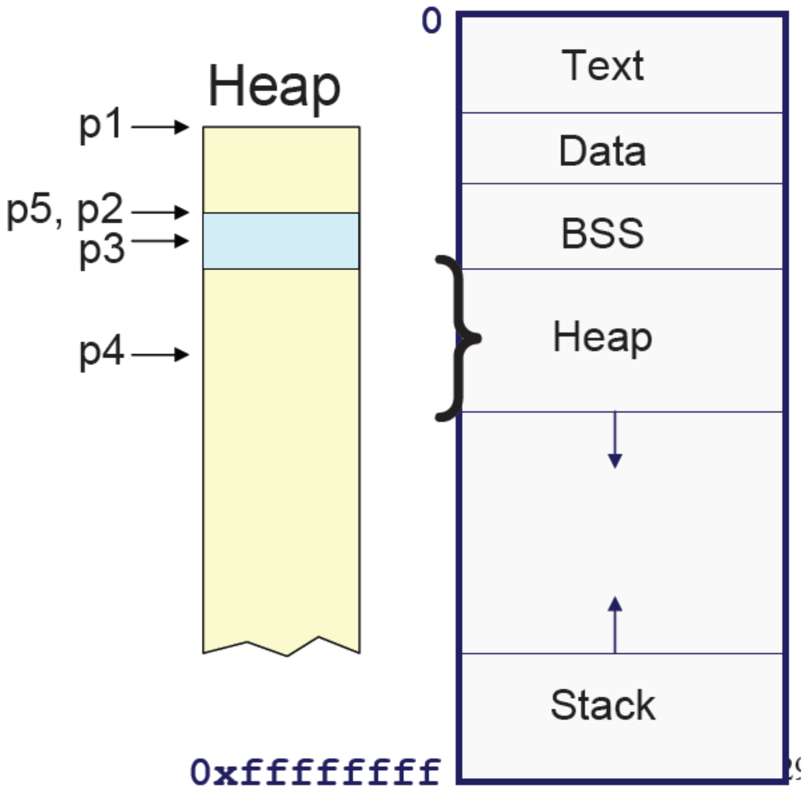
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

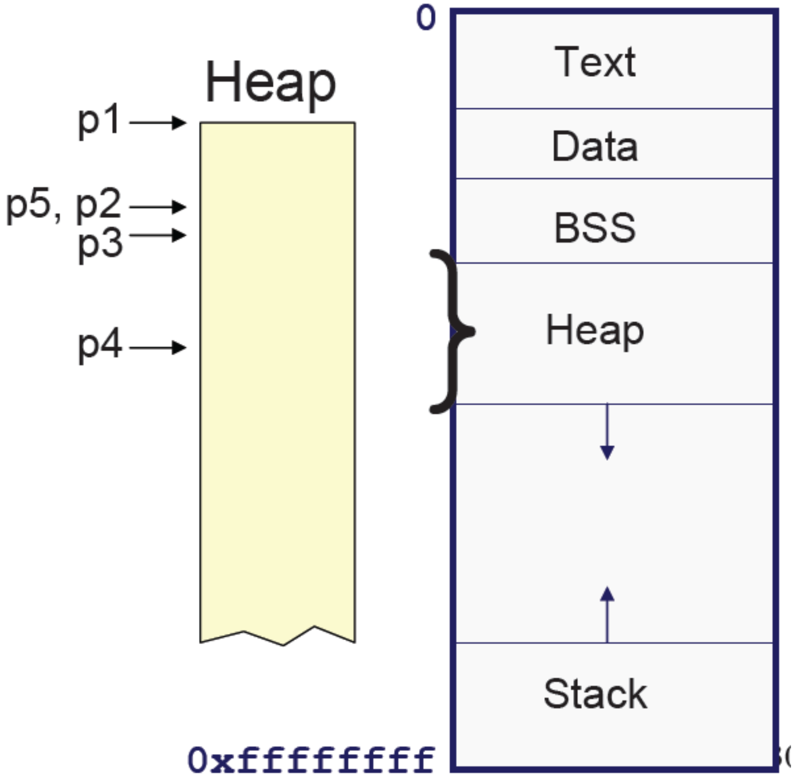
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Avoid Leaking Memory

- Memory leaks "lose" references to dynamic memory

```
int f (void)
{
        char * p;
        p = (char *) malloc (8 * sizeof(char));
        ….
        return 0;
}
int main (void) {
        f ( );
        …
}
```

Avoid Dangling Pointers

- Dangling pointers point to data that's not there anymore

```
char *f (void)
{
        char p[8];
        ….
        return p;
}
int main(void) {
        char *res = f ( );
}
```

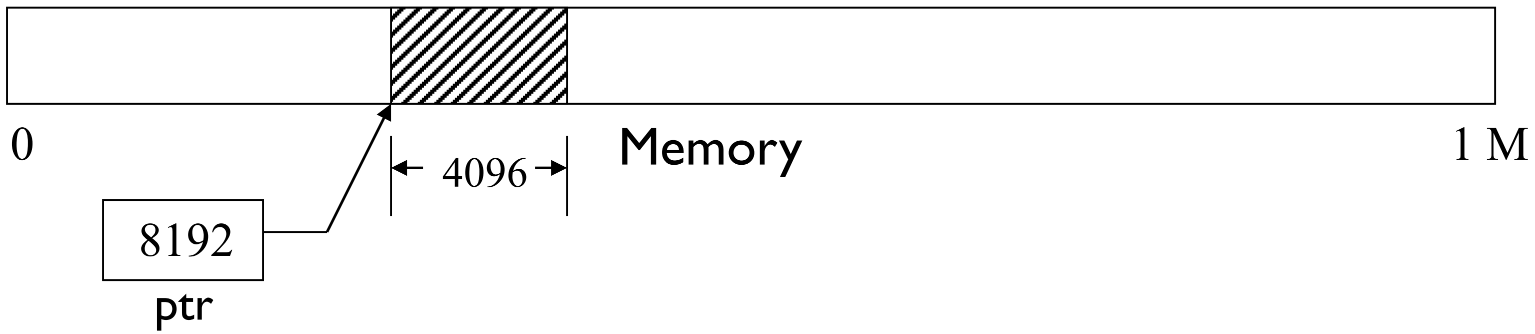
Memory Allocation

- Memory is requested and granted in contiguous blocks
  - Think of memory as one huge array of bytes
  - *malloc* library call
  - used to allocate memory
  - finds sufficient contiguous memory
  - reserves that memory
  - returns the address of the first byte of the memory
  - *free* library call
  - give address of the first byte of memory to free
  - memory becomes available for reallocation

- Same for allocating and freeing memory for processes in OS

Example of Memory Allocation

```
char* ptr = malloc(4096);        // char* is address of a
  single byte
```

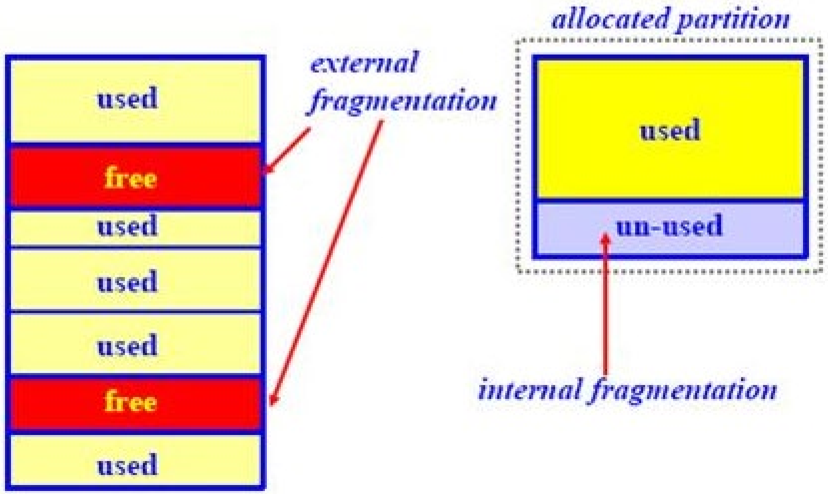


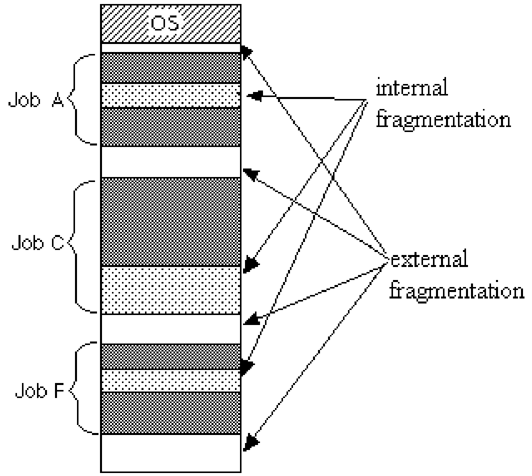
Fragmentation

- Segments of memory can become unusable after allocation and de-allocation

- External fragmentation
  - Variable allocation sizes
  - Memory remains unallocated

- Internal fragmentation
  - Fixed allocation sizes
  - Memory is allocated but unused

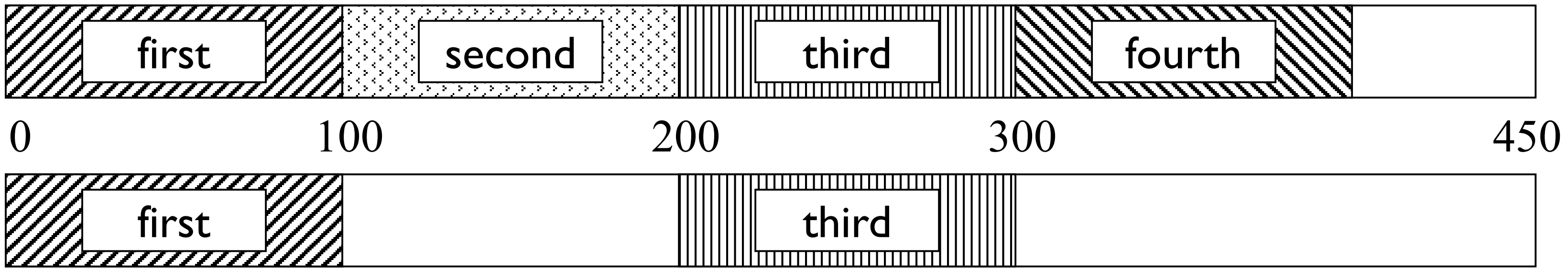




External Fragmentation

- A series of *malloc* and *free*

```
char* first = malloc(100);
char* second = malloc(100);
char* third = malloc(100);
char* fourth = malloc(100);
free(second);
free(fourth);
char* problem = malloc(200);
```

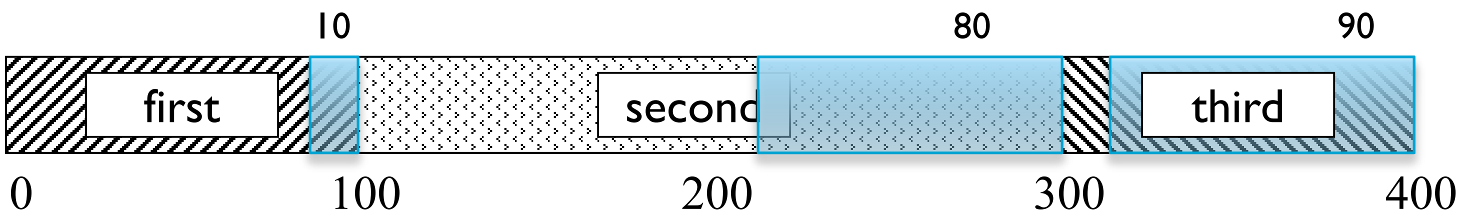


- 250 bytes of free memory, only 150 contiguous
  - unable to satisfy final malloc request `malloc(200)`

Internal Fragmentation

- A series of *malloc*
  - Assume allocation unit is 100 bytes

```
char* first = malloc(90);
char* second = malloc(120);
char* third = malloc(10);
char* problem = malloc(50);
```



- All of memory has been allocated but only a fraction of it is used (220 bytes)
  - unable to handle final memory request `malloc(50)`

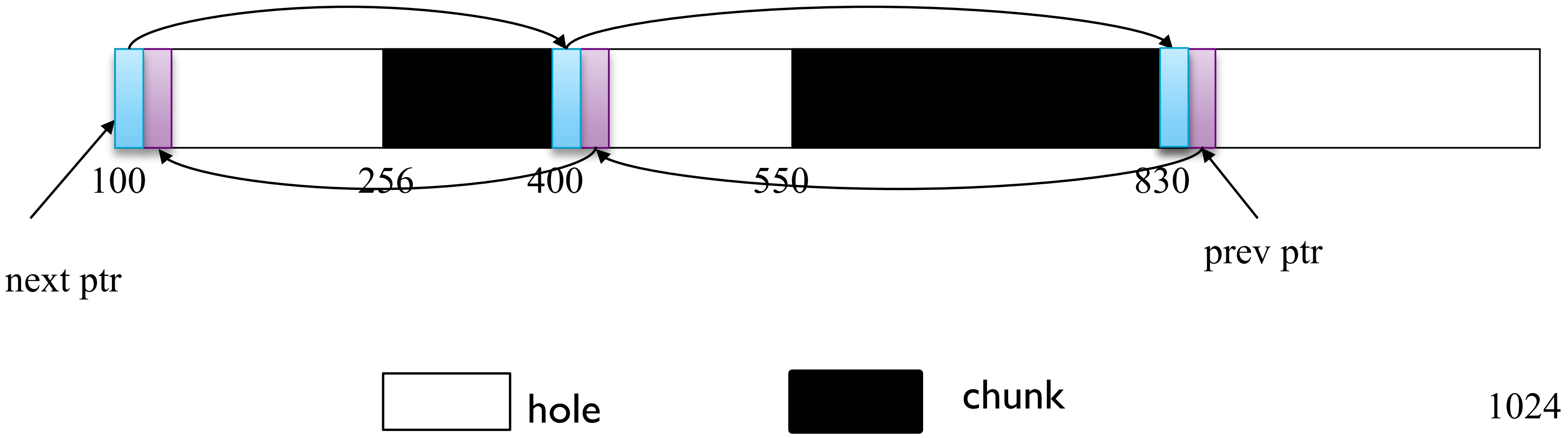
Internal vs. External Fragmentation

- Externally fragmented memory can be compacted (later)

- Fixed size allocation may lead to internal fragmentation, but less overhead to keep track of free memory
  - 8192 byte area of free memory
  - request for 8190 bytes
  - if exact size allocated: 2 bytes left to keep track of
  - if fixed size of 8192 used: 0 bytes to keep track of

Free List: External Fragmentation

- Need to keep track of available memory
  - Contiguous block of free mem is called a "hole"
  - Contiguous block of allocated mem is a "chunk"
- Keep a doubly linked list of free space
  - Build the pointers directly into the holes

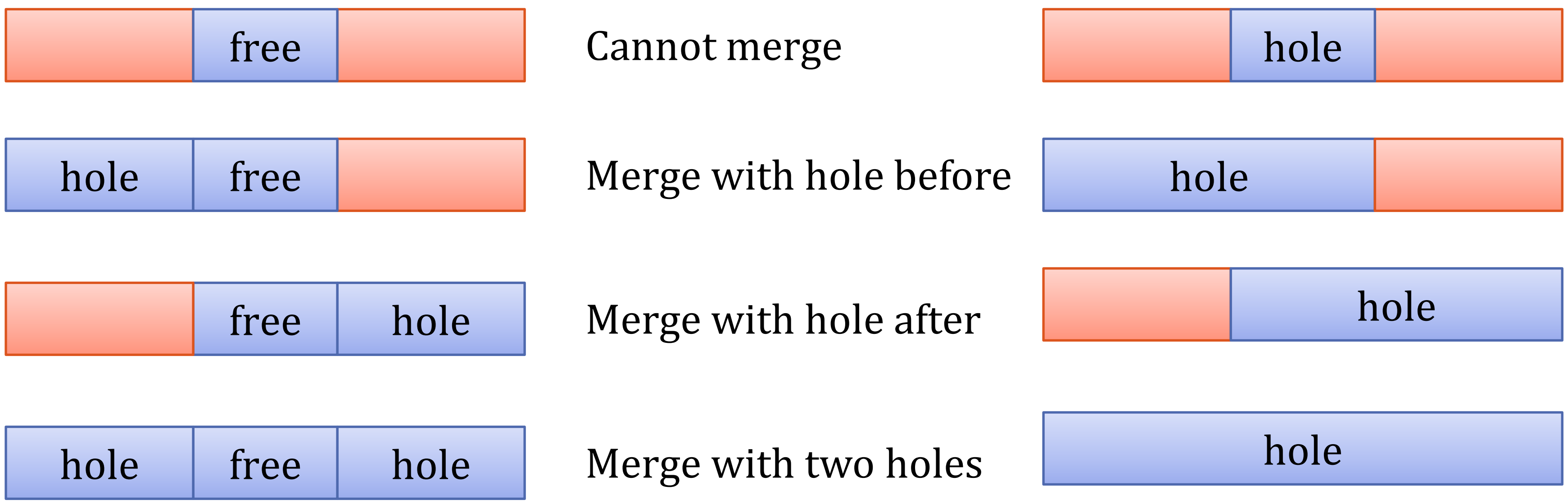


Free List

- Prefer holes to be as large as possible
  - Large hole can satisfy a small request
  - The opposite is not true
  - Less overhead in tracking memory
  - Fewer holes; so faster search for available memory

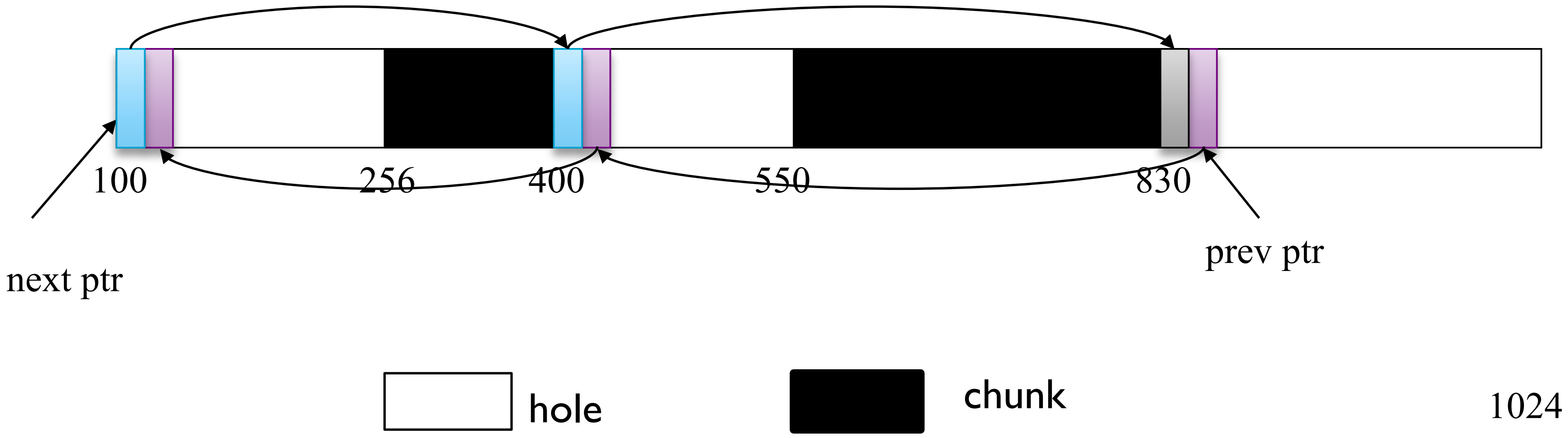
Deallocating Memory

- When memory is freed
  - Place memory in free list; set next and previous pointers
  - Merge with hole before and/or after if possible



Free List: Deallocating Memory

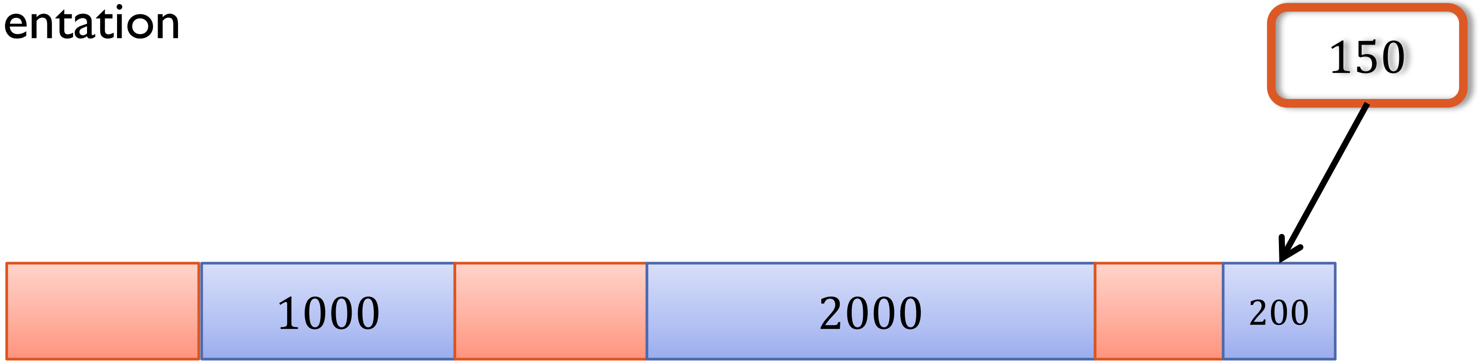
- Place memory in free list; set next and previous pointers
- Merge with hole before and/or after if possible



Allocation Algorithms: Best Fit

- Best Fit
  - Pick smallest hole that will satisfy the request

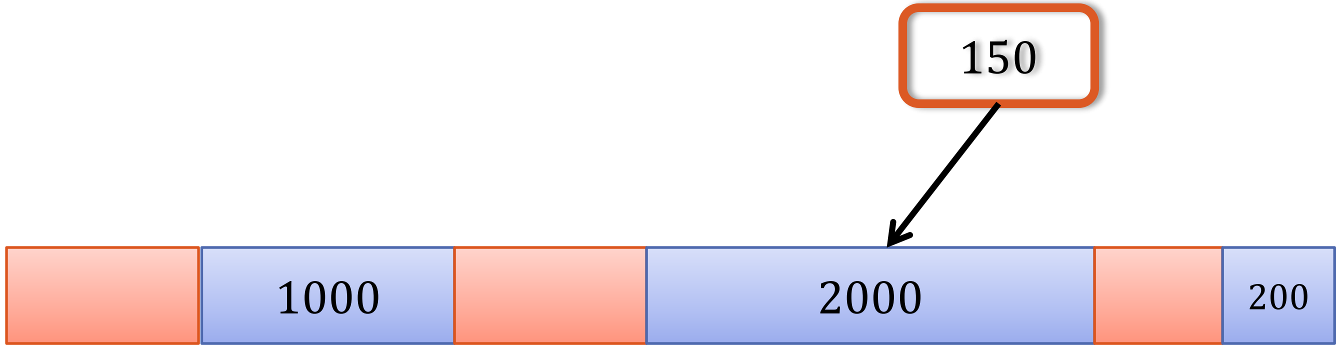
- Comments
  - Have to search entire list every time
  - Tends to leave lots of small holes
  - External fragmentation



Allocation Algorithms: Worst Fit

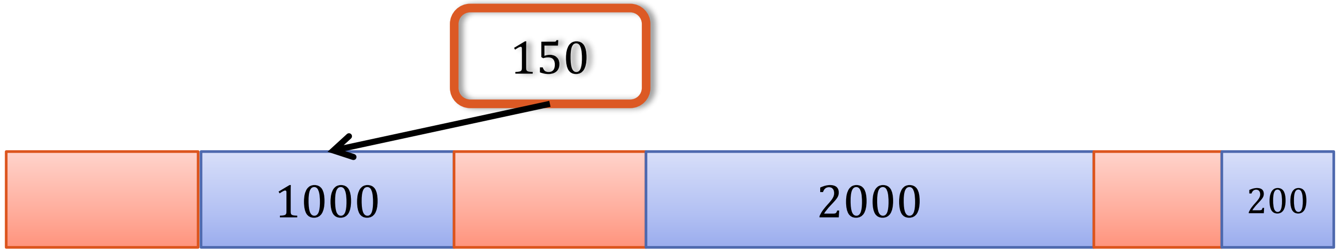
- Worst fit
  - Pick the largest hole to satisfy request

- Comments
  - Have to search entire list
  - Still leads to fragmentation issues



Allocation Algorithms: First Fit, Next Fit

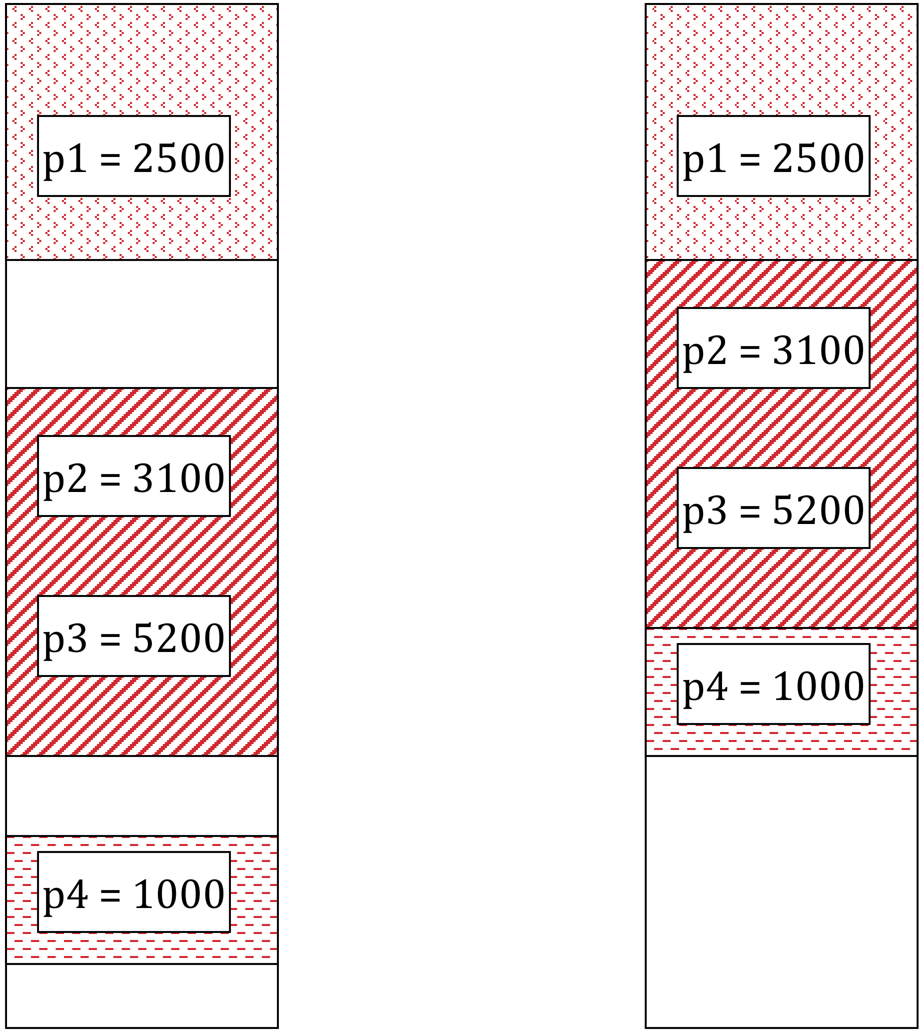
- First fit
  - Pick the first hole large enough to satisfy the request
- Comments
  - Much faster than best or worst fit
  - Has fragmentation issues similar to best fit
- Next fit
  - Exactly like first fit except start search from where last search left off



Compaction

- To deal with internal fragmentation
  - Use paging or segmentation
  - More on this in next week's lecture
- To deal with external fragmentation
  - Can do compaction

Compaction



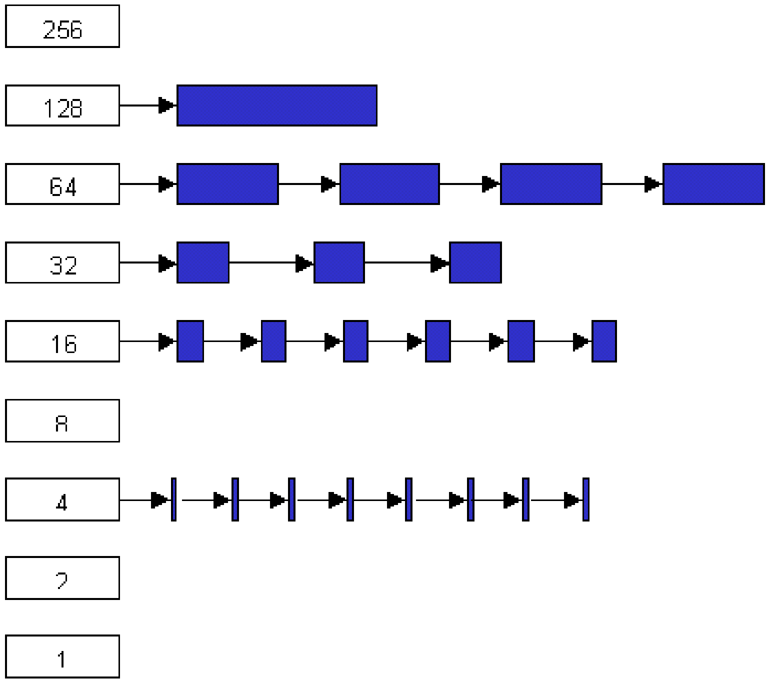
Compaction

- Simple concept
  - Move all allocated memory locations to one end
  - Combine all holes on the other end to form one large hole

- Major problems
  - Tremendous overhead to copy all data
  - Must find and change all pointer values
  - This can be very difficult

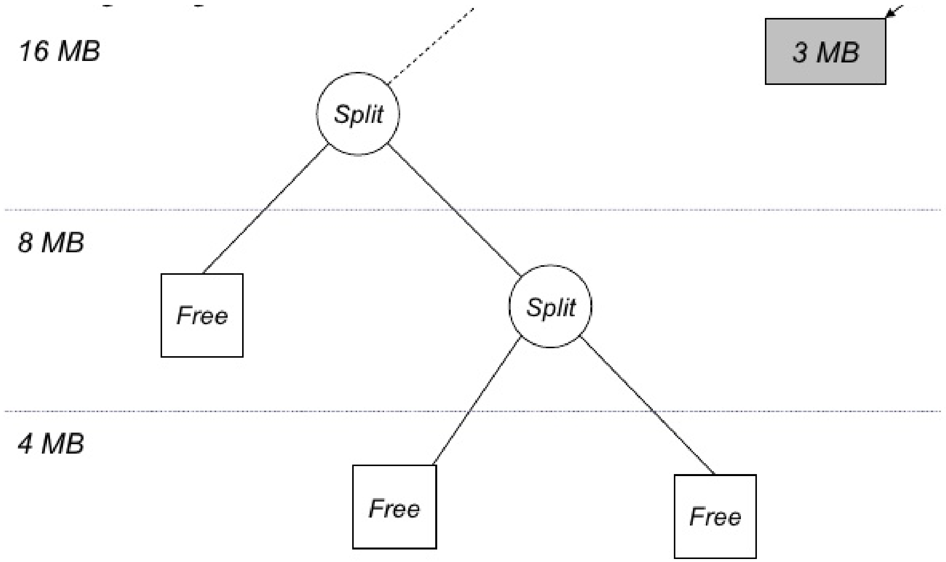
Multiple Free Lists

- Keep multiple lists of different hole sizes
- Take hole from a list that most closely matches size of request
- Leads to internal fragmentation
  - ~50% of memory in the common case



Multiple Free Lists

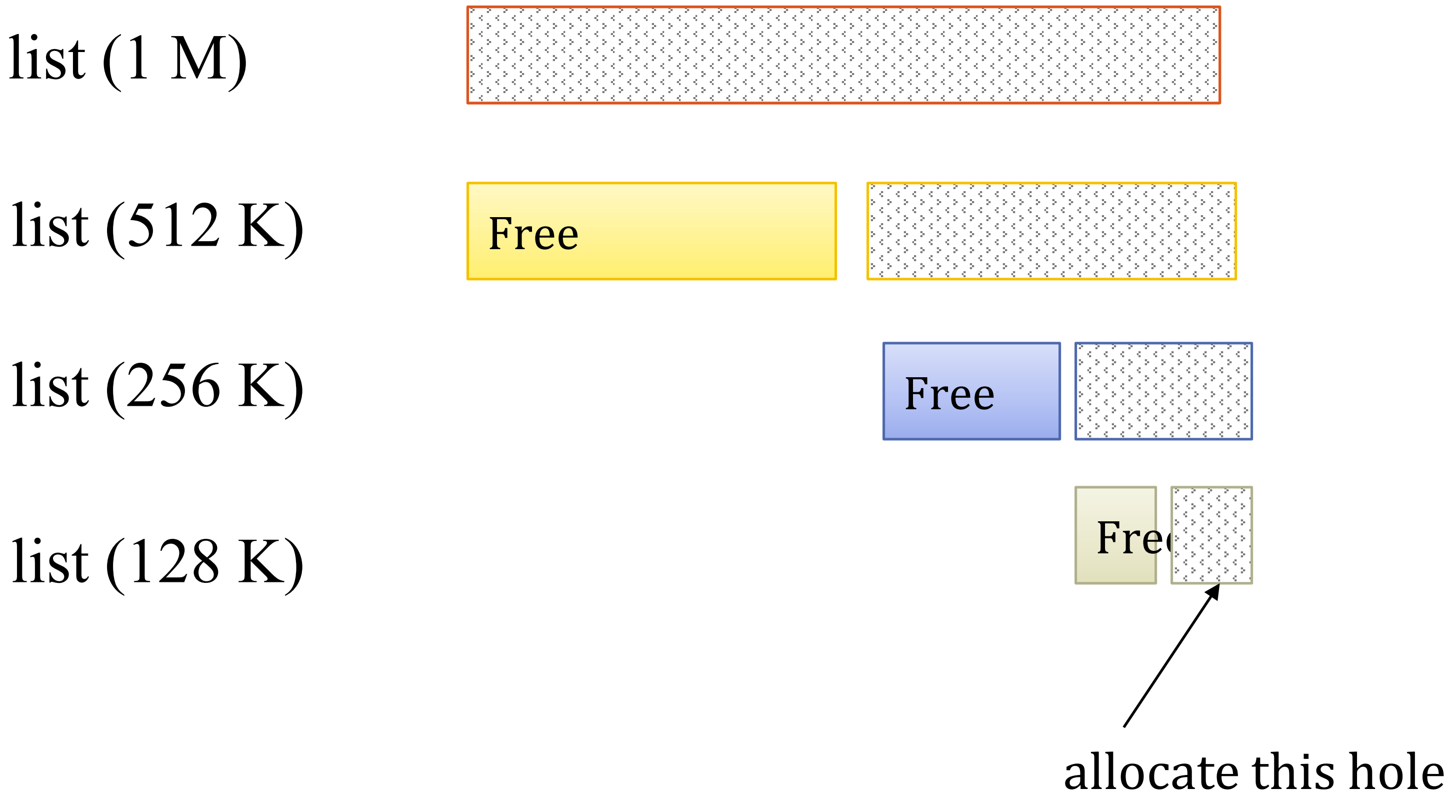
- Start out with single large hole
  - One entry in one list; Hole size is usually a power of 2
- Upon a request for memory, keep dividing hole by 2 until appropriate size memory is reached
  - At every division, a new hole is added to a different free list
- One a hole is created, it cannot merge with another hole



Multiple Free Lists

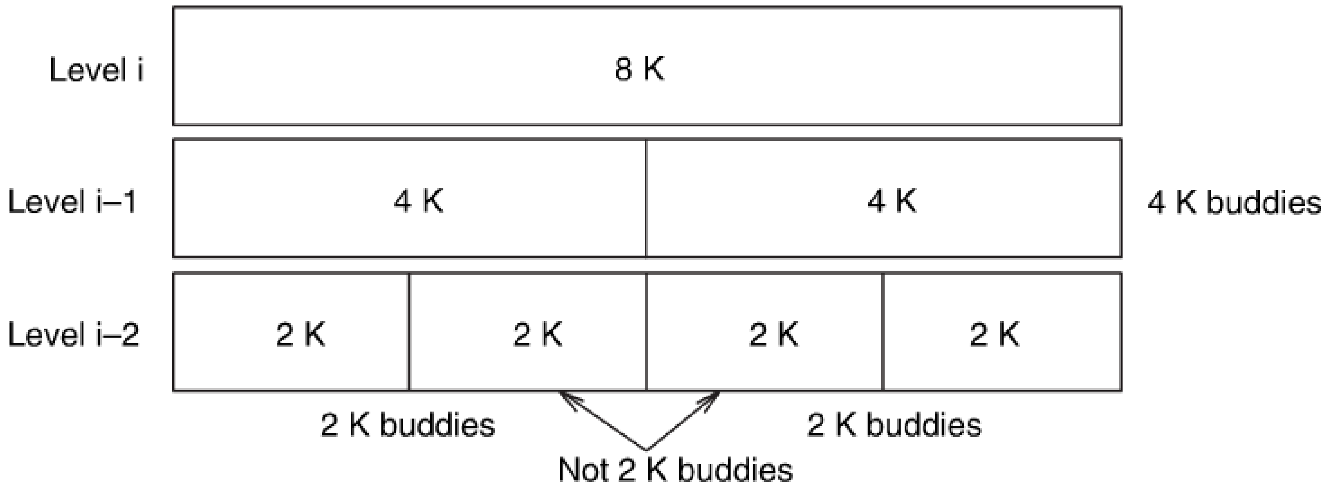
```
char* ptr = malloc(100 K)
```

- Initially, only one entry in first list (1 M)
- In the end, one entry in each list except 1 M



Buddy System

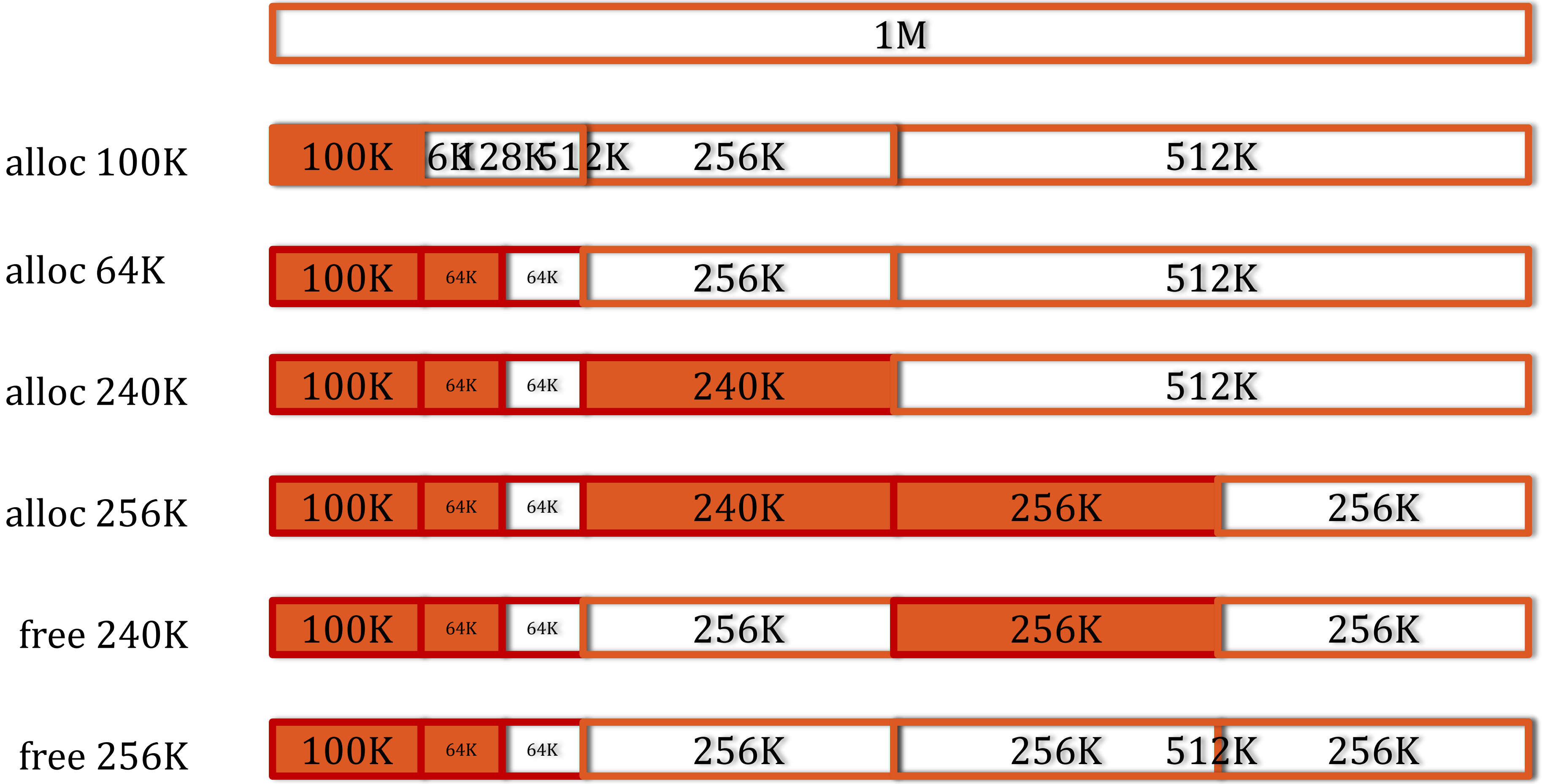
- Each hole in a free list has a *buddy*
  - If a hole and its buddy are combined, they create a new hole
  - New hole is twice as big and is aligned on proper boundary
- Example
  - A hole is of size 4
  - Starting location of each hole: 0, 4, 8, 12, 16, 20, …
  - Buddies are the following: (0,4), (8, 12), …
  - If buddies are combined, get holes of size 8
  - starting location of these holes: 0, 8, 16, ...



Buddy System

- When allocating memory
  - If list is empty, go up one level, take a hole and break it in 2
  - These 2 new holes are buddies
  - Now give one of these holes to the user
- When freeing memory
  - If chunk just returned and its buddy are in the free list, merge them and move the new hole up one level

Buddy System Example



Summary

- In this lecture, we learnt about different components of process memory
- We also learnt simple memory allocation and deallocation schemes