CS61C Midterm Review on C & Memory Management

Fall 2006

Aaron Staley

Some material taken from slides by:

Michael Le

Navtej Sadhal

Overview

- C
  - Array and Pointer Goodness!
- Memory Management

  The Three Three's!

Pointers in C

- Pointers
  - A pointer contains an address of a piece of data.
  - & gets the address of a variable
  - * dereferences a pointer

```
int a; /*Declare a*/
int *b = &a; /*get address of A*/
int c = *b; /*dereference B - get C*/
```

Pointer Math

- Consider

```
int * a = malloc(3*sizeof(int));
int * b = a + 2;
```

- This is the same as:

```
int * a = malloc(3*sizeof(int));
int * b = (int*)( ((int)a) + 2*sizeof(*a));
```

(In other words, b will increase by 8 in this example)

Arrays in C

- Arrays vs. Pointers

  -Interchangeable when used in a function:

  ```
  void foo (int * a); IS
       void foo (int a[]);
  ```

  -array[index] is equivalent to *(array+index)

  b[i]; IS /*remember pointer math!*/

  *(b+i);

  -Arrays also have a special declaration to allocate stack space.

  int c[5]; /*creates 5 integers on the stack*/

  NOTE: c acts like a special “read-only pointer” that can’t be modified!

A Note about C Declarations

- Declarations have same syntax as use!
  -int * a[2]; /*declare this*/
  -Now doing *a[1] will return an int!

Question:

Is int * a[2] declaring an array of 2 pointers to integers or a pointer to arrays of 2 integers?

A Note about C Declarations

- Declarations have same syntax as use!
  - -int * a[2]; /*declare this*/
  - -Now doing *a[1] will return an int!

Question:

Is int * a[2] declaring an array of 2 pointers to integers or a pointer to arrays of 2 integers?

IT IS AN ARRAY OF 2 POINTERS TO INTEGERS!
This is because a[1] would return an int*!

And Structures/Unions

- Struct keyword defines new datatypes:

```
struct binTree{
    int a;
    struct binTree * left;
    struct binTree * right;
};
```

`So: sizeof(struct binTree) == 12`

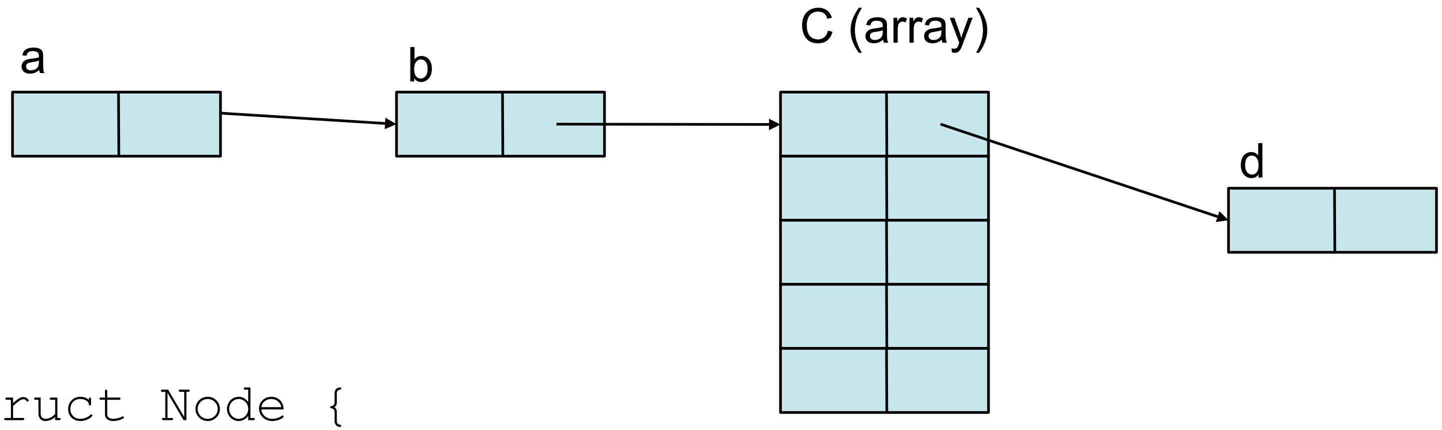
- Unions allow types to be used interchangeably. Fields all use the same memory. Size is the largest field:

```
union anything{
    char a;
    int b;
    void * c;
};
```

So: sizeof (union anything) == 4

Pointers

How would you create this situation in C without using malloc()?




```
struct Node {
      int i;
      struct Node * next;
};
```

Pointers

```
struct Node {
   int i;
   struct Node * next;
};

int main() {
   struct Node a, b, c[5], d;
   a.next = &b;
   b.next = c;
   c[0].next = &d;  /* c->next =&d; is also valid*/
   return 0;
}
```

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How do you allocate an array of 10 integers?

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How do you allocate an array of 10 integers?

```
int *i= malloc(sizeof(int)*10);
```

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How do you allocate an array of 10 integers?

```
int *i= malloc(sizeof(int)*10);
```

- String of length 80?

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How do you allocate an array of 10 integers?

```
int *i= malloc(sizeof(int)*10);
```

- String of length 80?

```
char *str= malloc(sizeof(char)*81);
```

/*Remember: Strings end with '\0'*/

Malloc

- Allocates memory on the heap
- Data not disappear after function is removed from stack
- How do you allocate an array of 10 integers?

```
int *i= malloc(sizeof(int)*10);
```

- String of length 80?

```
char *str= malloc(sizeof(char)*81);
```

- If you don't free what you allocate, you memory leak:

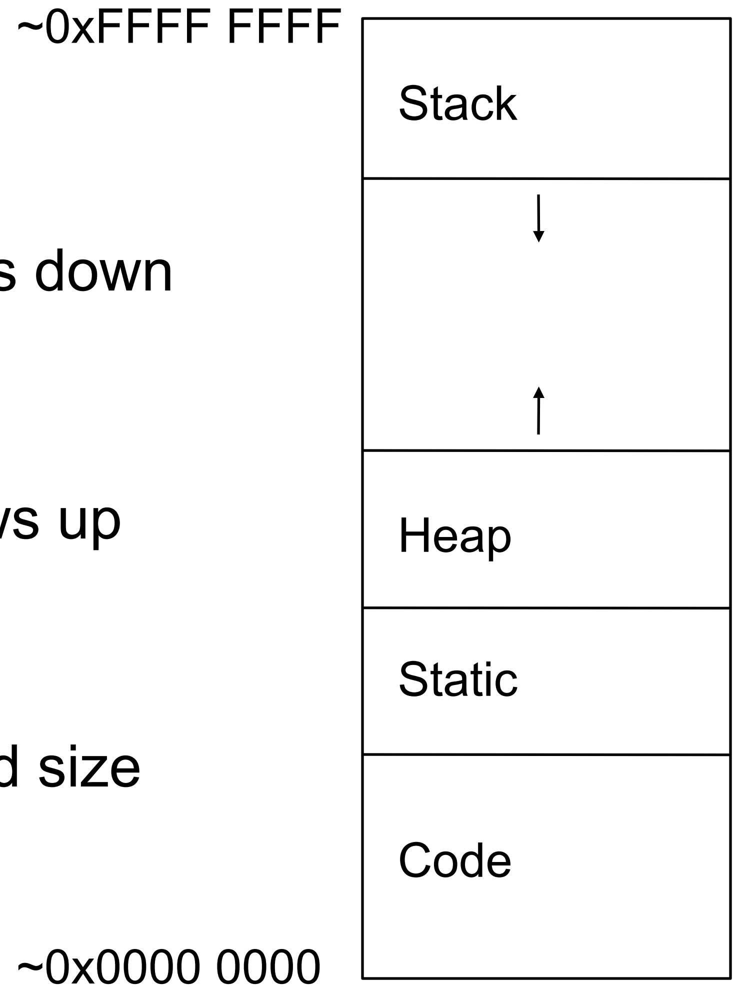
```
free (str); /*Do this when done with str*/
```

Memory Management

Stack: local variables, grows down (lower addresses)

Heap: malloc and free, grows up (higher addresses)

Static: global variables, fixed size



Pointers & Memory

You have a linked list which holds some value.

You want to insert in new nodes in before all nodes of a certain value.

```
struct node {
   int * i; /*pointer to some value in STATIC memory*/
   struct node * next; /*next*/
};
typedef struct node Node; /*typedef is for aliasing!*/


void insertNodes(Node **lstPtr, int oldval, /
   *something*/) {
   …
}
```

Pointers & Memory

```
struct node {
   int * i; /*pointer to some value in STATIC
                      memory*/
   struct node * next; /*next*/
};
typedef struct node Node;

/*NOTE: lstPtr is a handle here.  We do this in case
   the HEAD of the list is removed!*/
```

Which is correct?

```
void insertNodes(Node **lstPtr, int oldVal, int * newVal)
OR
void insertNodes(Node **lstPtr, int oldVal, int newVal)
```

Pointers & Memory

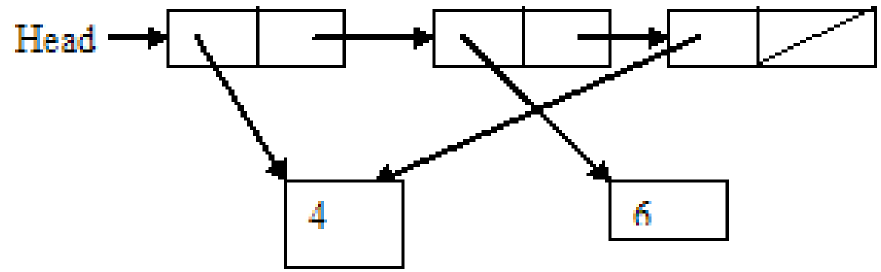
```
struct node {
   int * i; /*pointer to some value in STATIC
                     memory*/
   struct node * next; /*next*/
};
typedef struct node Node;

/*NOTE: lstPtr is a handle here.  We do this in case
   the HEAD of the list is removed!*/
```

Which is correct?

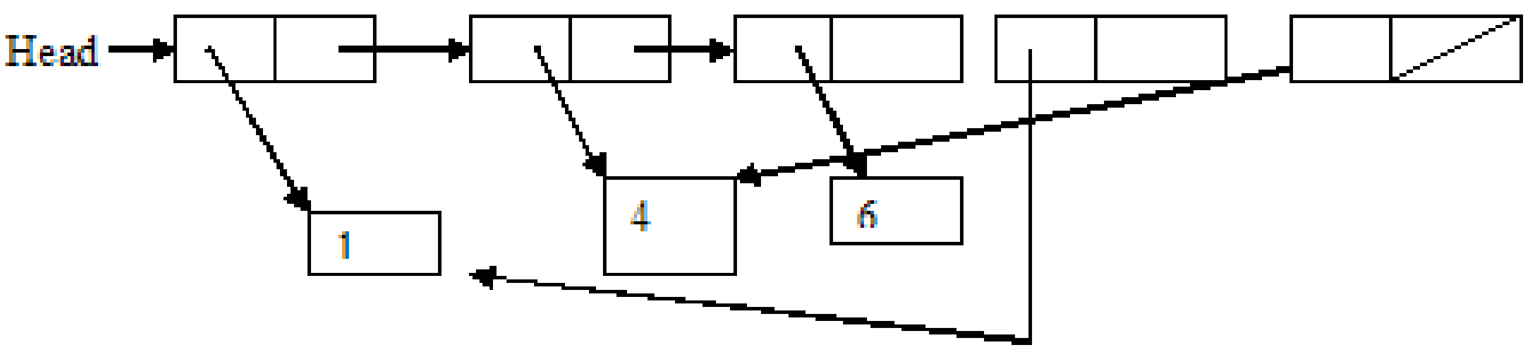
```
void insertNodes(Node **lstPtr, int oldVal, int * newVal)
OR
void insertNodes(Node **lstPtr, int oldVal, int newVal)
```

In Pictures

- List looks like:



- `insertNodes(&head, 1, ptr_to_1);`
  - Has no effect
- `insertNodes(&head, 4, ptr_to_1);`
  - List becomes:



  - A small hint: &(f.a) will return address of field a (of structure f)

Pointers

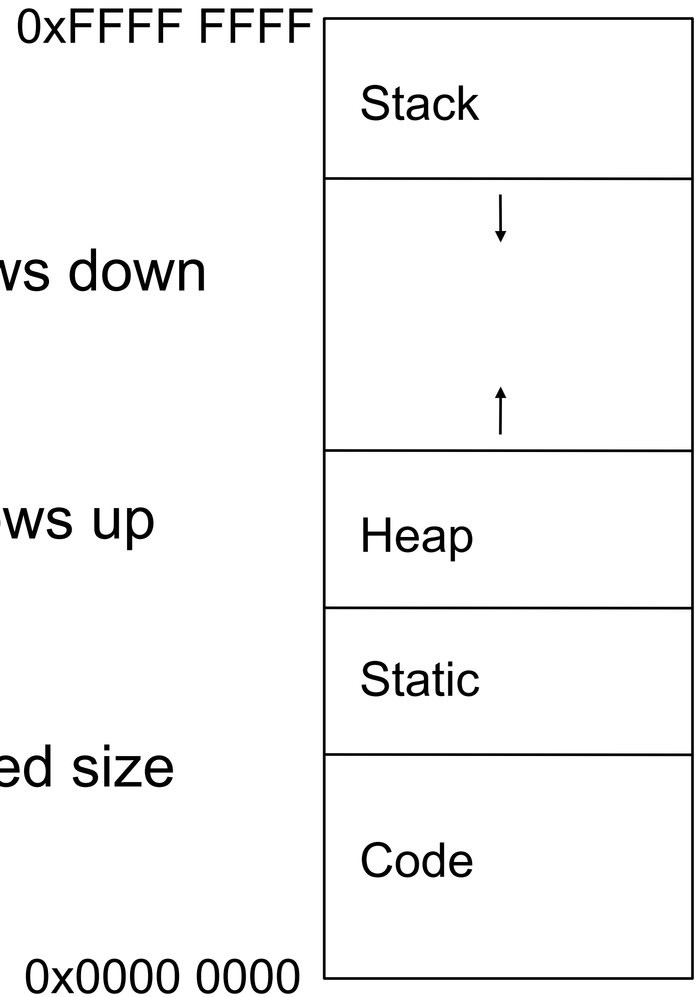
```
void insertNodes(Node **lstPtr, int oldVal, int * newVal)
      if ((*lstPtr)==NULL) {
            /*Base CASE*/
      } else if (*((*lstPtr)->i) == oldVal) {
            /*Equality*/
            /*Insert before this node*/
            /*Update *lstPtr somehow?*/
      } else {
            /*Inequality.. Resume*/
            /*But be careful with lstPtr!*/
      }
}
/*Recall that f->a IS (*f).a   */
```

Pointers

```
void insertNodes(Node **lstPtr, int oldVal, int * newVal)
      if ((*lstPtr)==NULL) {
            return;
      } else if (*((*lstPtr)->i) == oldVal) {
            Node * old = *lstPtr;
            *lstPtr = malloc(sizeof(Node));
            (*lstPtr)->i = newVal;
            (*lstPtr)->next = old;
            insertNodes (&(old->next),
                              oldVal,newVal);
      } else {
            insertNodes(
                  &((*lstPtr)->next),oldVal,newVal);
      }
}
```

A Reminder: Memory Management

- Stack: local variables, grows down (lower addresses)
- Heap: malloc and free, grows up (higher addresses)
- Static: global variables, fixed size



Memory (Heap) Management

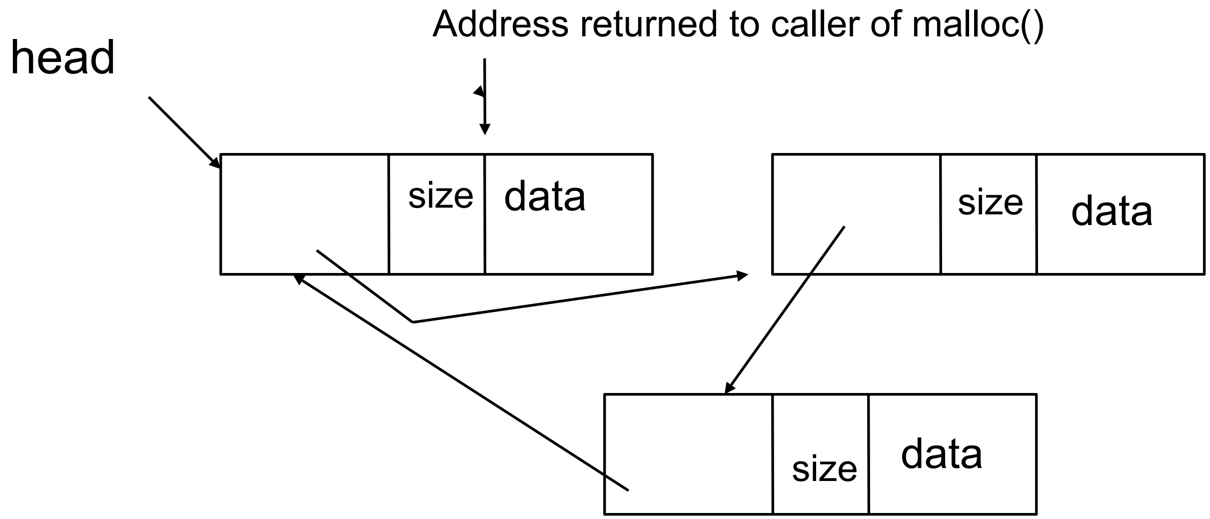
- When allocating and freeing memory on the heap, we need a way to manage free blocks of memory.

- Lecture covered three different ways to manage free blocks of memory.
    - Free List (first fit, next fit, best fit)
    - Slab Allocator
    - Buddy System

Free List

- Maintains blocks in a (circular) list:

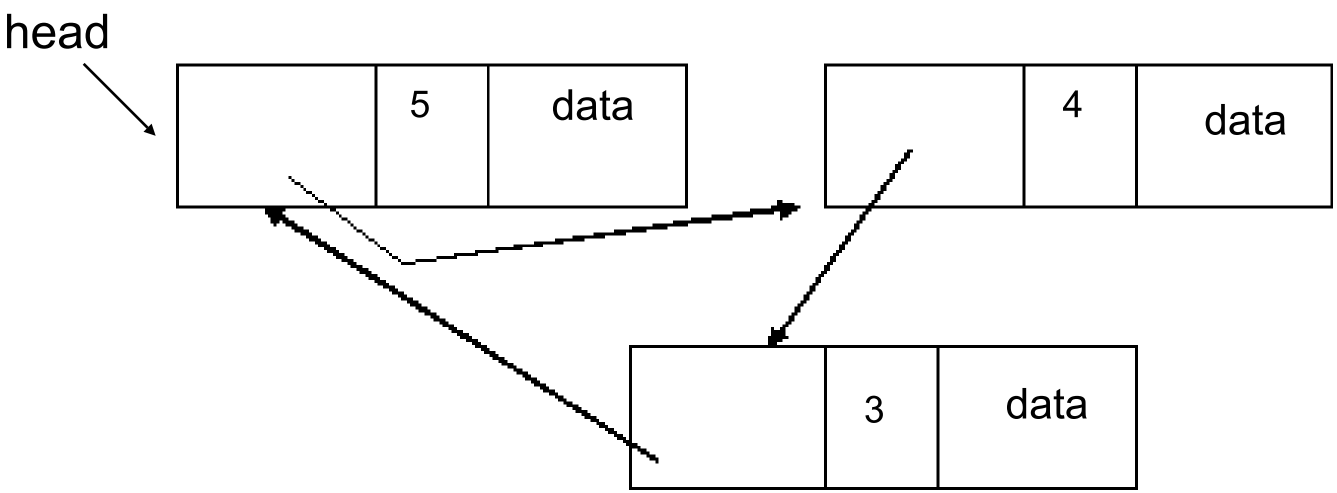
```
struct malloc_block{
  struct malloc_block * next;
  int size;
  uint8_t data[size]; /*WARNING: This is pseudocode*/
};
```



Free List Fits

- First Fit Selects first block (from head) that fits!
- Can lead to much fragmentation, but better locality (you'll learn why this is important)

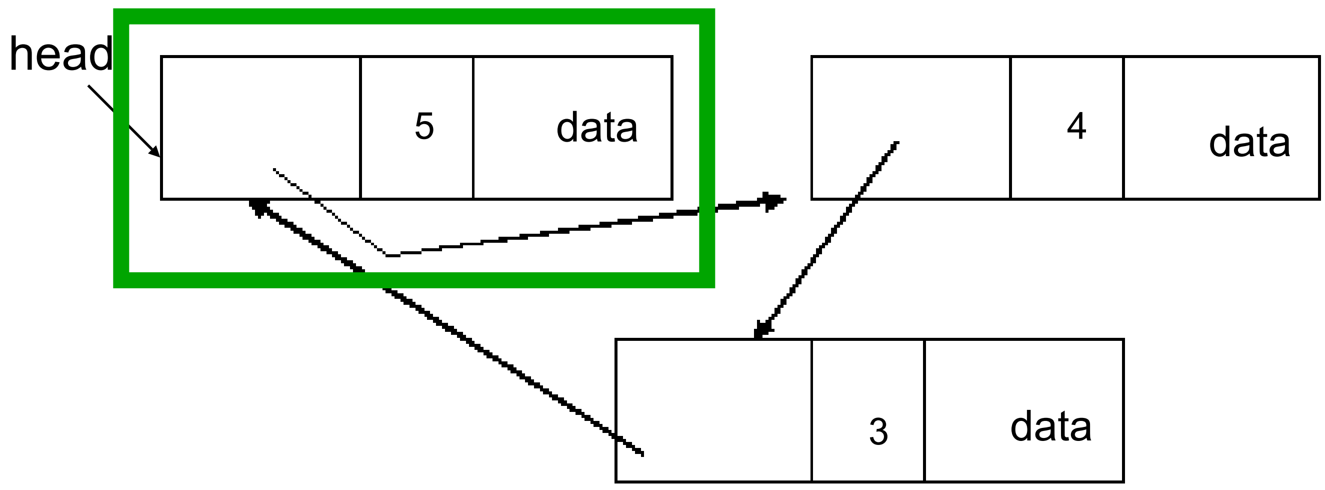
Example: `malloc (4*sizeof(char));`



Free List Fits

- First Fit Selects first block (from head) that fits!
- Can lead to much fragmentation, but better locality (you'll learn why this is important)

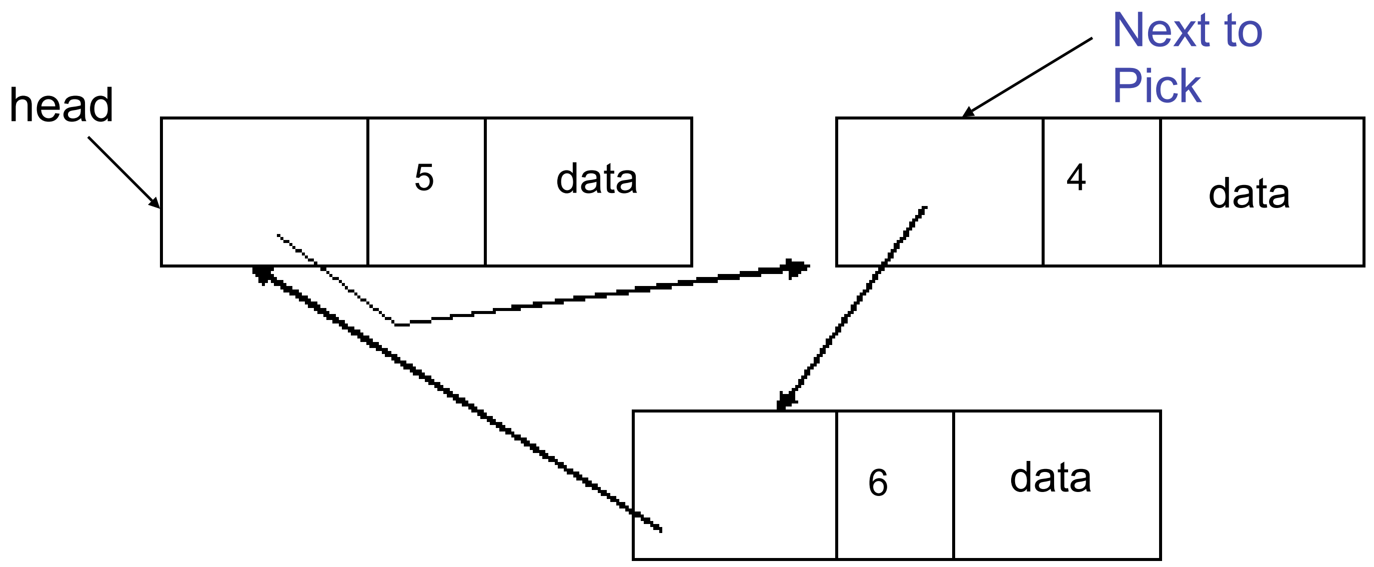
Example: `malloc (4*sizeof(char));`



Free List Fits

- Next Fit selects next block (after last one picked) that fits!
- Tends to be rather fast (small blocks everywhere!)

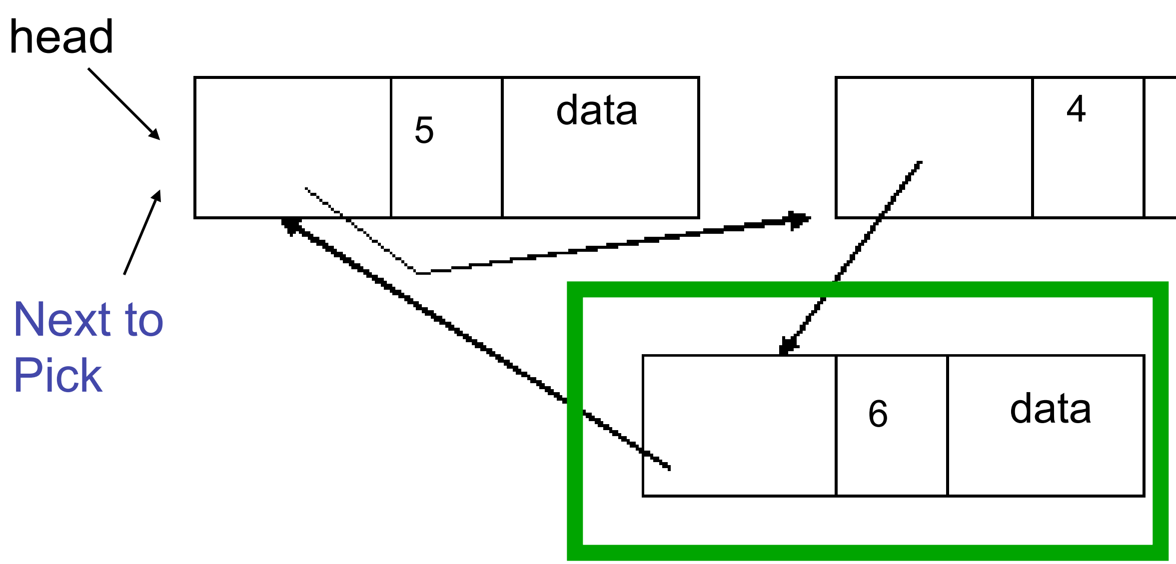
Example: `malloc (5*sizeof(char));`



Free List Fits

- Next Fit selects next block (after last one picked) that fits!
- Tends to be rather fast (small blocks everywhere!) =

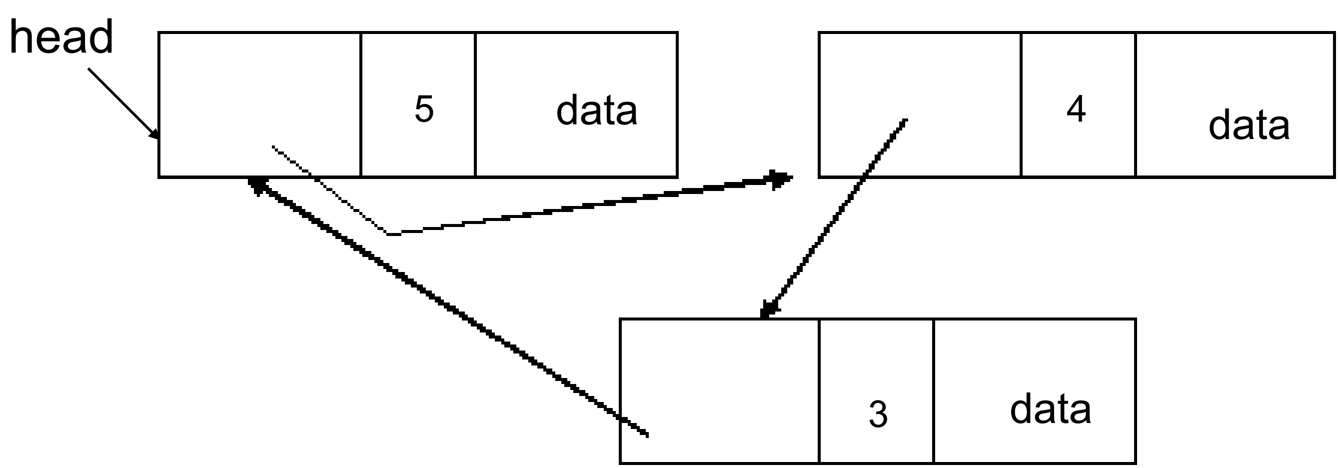
Example: `malloc (5*sizeof(char));`



Free List Fits

- Best fit picks the smallest block >= requested size.
- Tries to limit fragmentation, but can be slow (often searches entire list)!

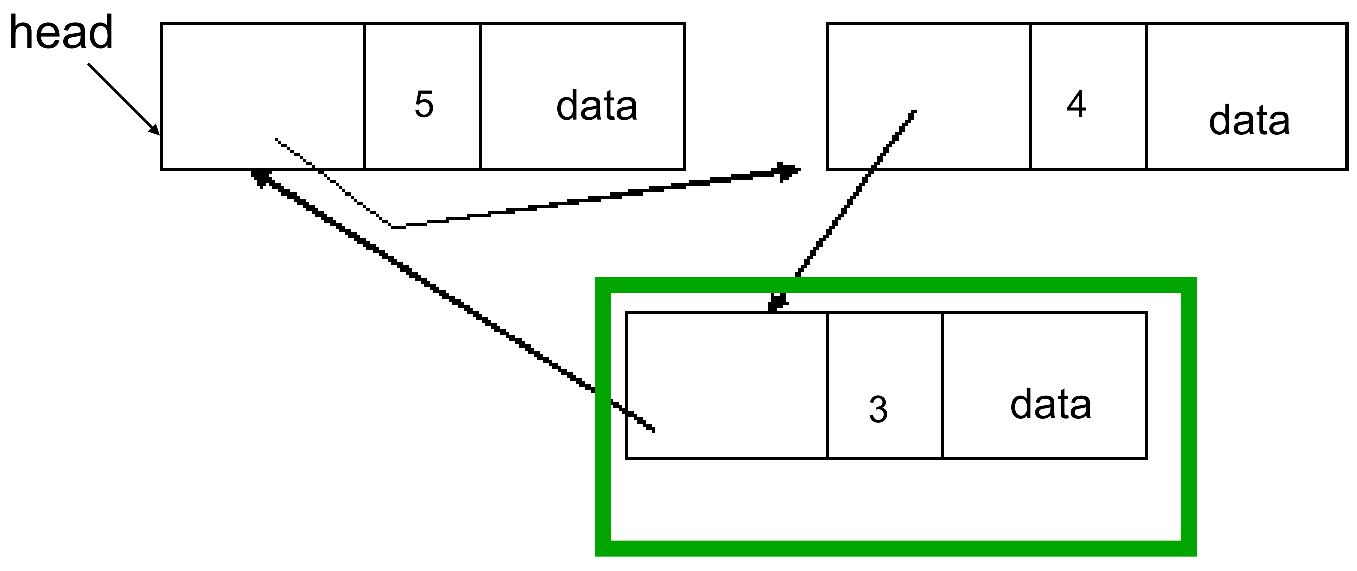
Example: `malloc (2*sizeof(char));`



Free List Fits

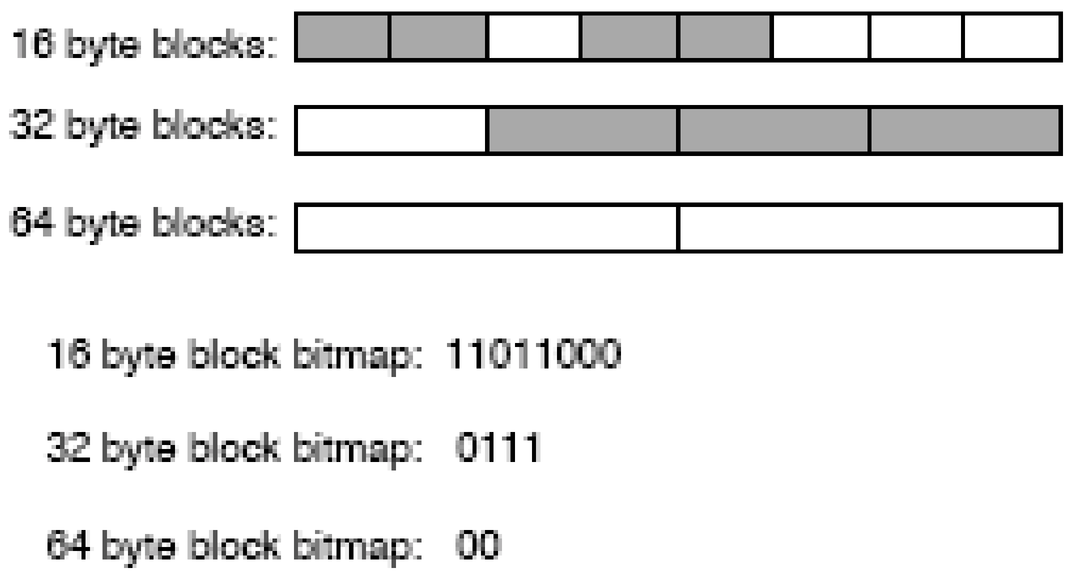
- Best fit picks the smallest block >= requested size.
- Tries to limit fragmentation, but can be slow (often searches entire list)!

Example: `malloc (2*sizeof(char));`



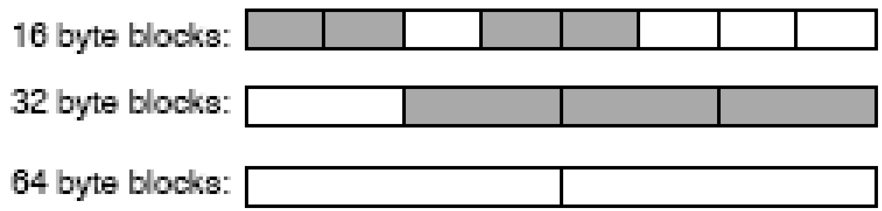
The Slab Allocator

- Only give out memory in powers of 2.
- Keep different memory pools for different powers of 2.
- Manage memory pool with bitmaps
- Revert to free list for large blocks.



The Slab Allocator

- Example: malloc(24*sizeof(char));
- Old:



16 byte block bitmap: 11011000

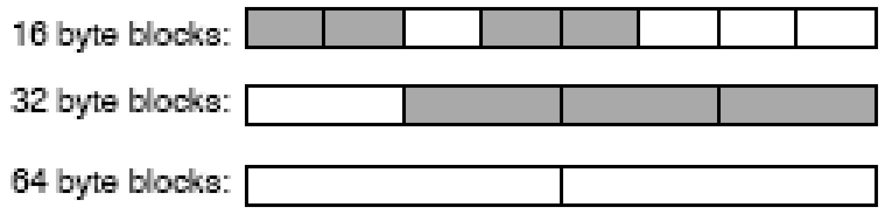
32 byte block bitmap: 0111

64 byte block bitmap: 00

New:

The Slab Allocator

- Example: malloc(24*sizeof(char));
- Old:

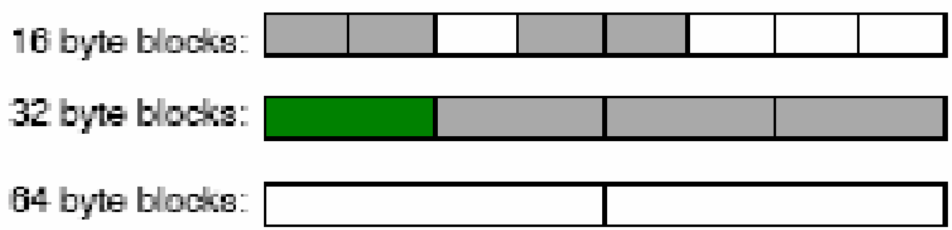


16 byte block bitmap: 11011000

32 byte block bitmap: 0111

64 byte block bitmap: 00

New:



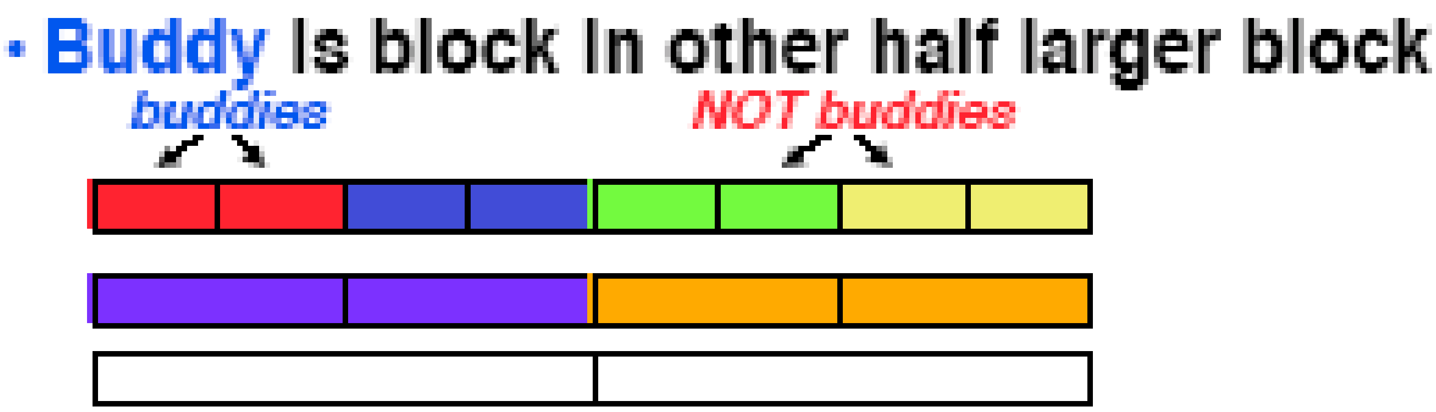
16 byte block bitmap: 11011000

32 byte block bitmap: 1111

64 byte block bitmap: 00

The Buddy System

- An adaptive Slab Allocator
- Return blocks of size n as usual.
- If not found, find block of size 2*n and split the block (This is recursive)!
- When block of size n is freed, merge it with its neighbor (if the neighbor is freed) into a block of size 2n (recursive!)



The Buddy System

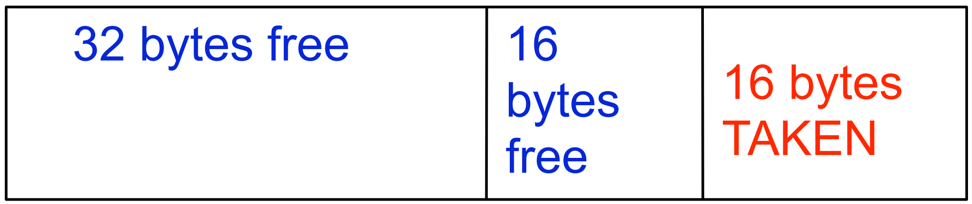
- Example:

32 bytes free	16 bytes free	16 bytes TAKEN

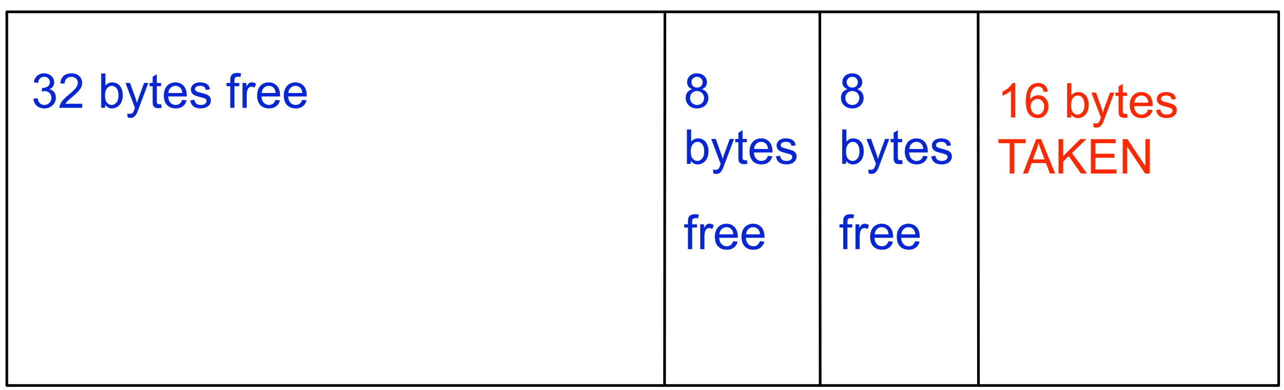
malloc(7*sizeof(char)); /*force request to be 8 bytes*/

The Buddy System

- Example:

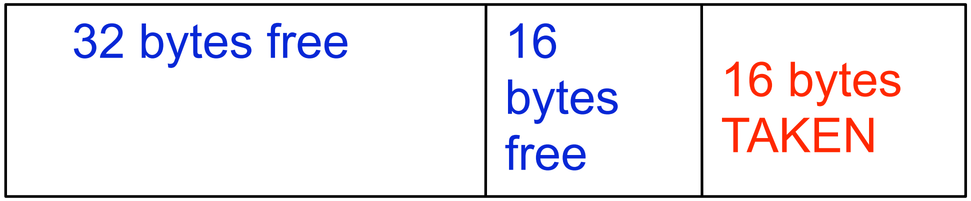


malloc(7*sizeof(char)); /*force request to be 8 bytes*/

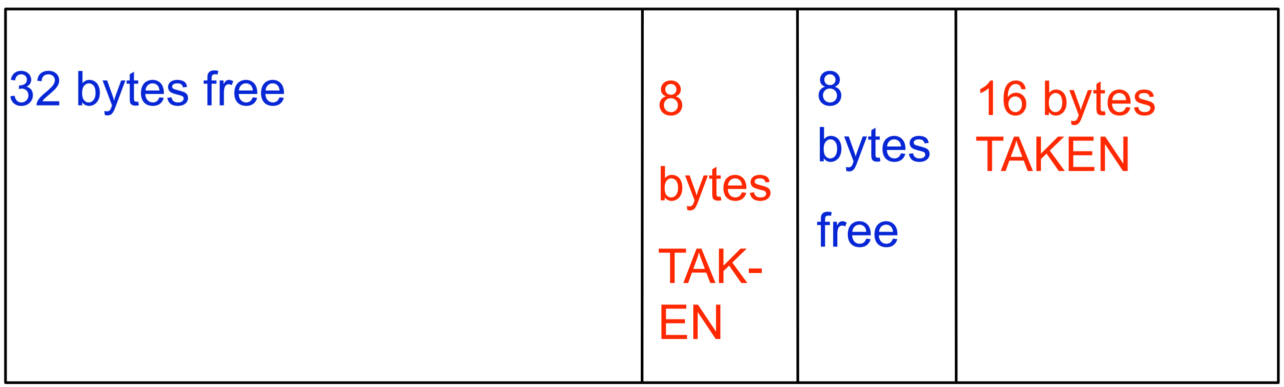


The Buddy System

- Example:

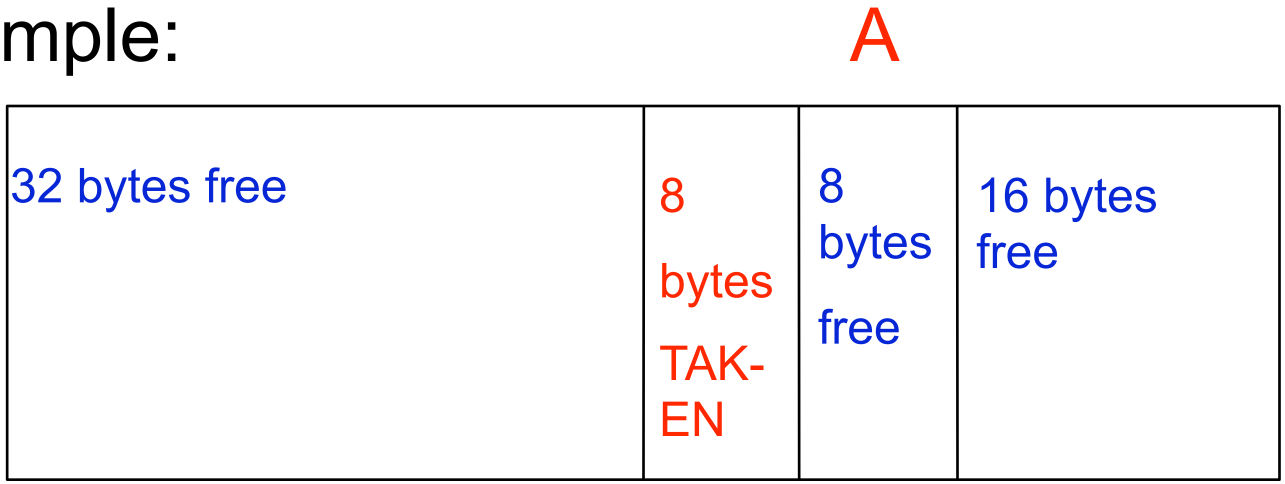


malloc(7*sizeof(char)); /*force request to be 8 bytes*/



The Buddy System

- Example:



free (a);

The Buddy System

- Example:

A

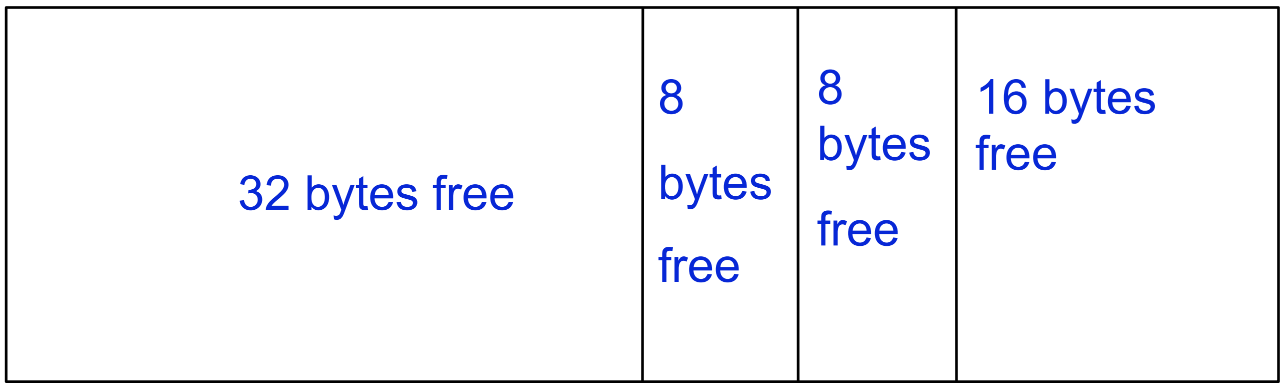
32 bytes free	8 bytes TAK- EN	8 bytes free	16 bytes free

free (a);

32 bytes free	8 bytes free	8 bytes free	16 bytes free

The Buddy System

- Example: free (a);

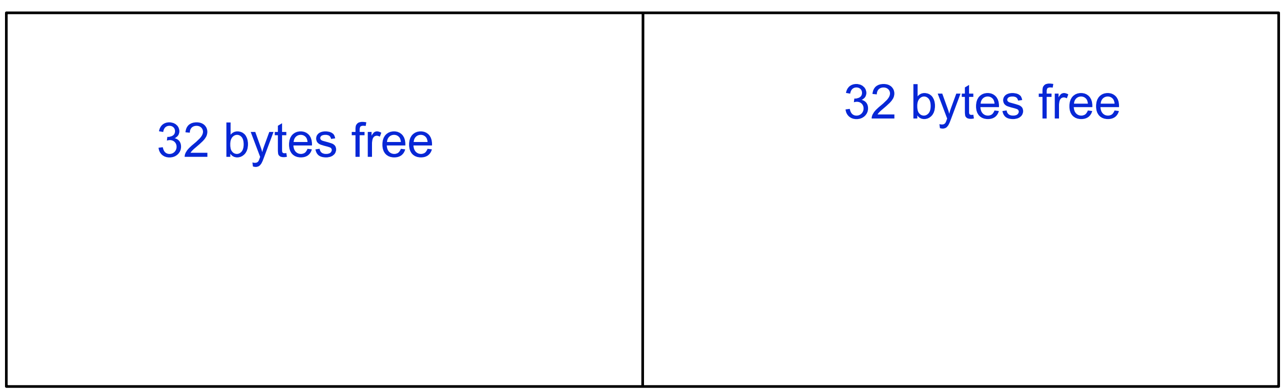


- Coalesce:

The Buddy System

- Example: free (a);

- Coalesce:



The Buddy System

- Example: free (a);

32 bytes free | 32 bytes free

- Coalesce:

64 bytes free =)

A Word about Fragmentation

- Internal fragmentation: Wasted space within an allocated block (i.e. I request 30 bytes but get a 32 byte block back)
- External Fragmentation: Wasted space between allocated blocks (if blocks were compacted, we could have more *contiguous* memory)

☺ An Old Midterm Question ☺

For each of the allocation systems on the left, circle the column that describes its *fragmentation*:

Buddy System	Causes *internal* only	Causes *external* only	Causes *both types*
Slab Allocator	Causes *internal* only	Causes *external* only	Causes *both types*
K&R (Free List Only)	Causes *internal* only	Causes *external* only	Causes *both types*

☺ An Old Midterm Question ☺

For each of the allocation systems on the left, circle the column that describes its *fragmentation*:

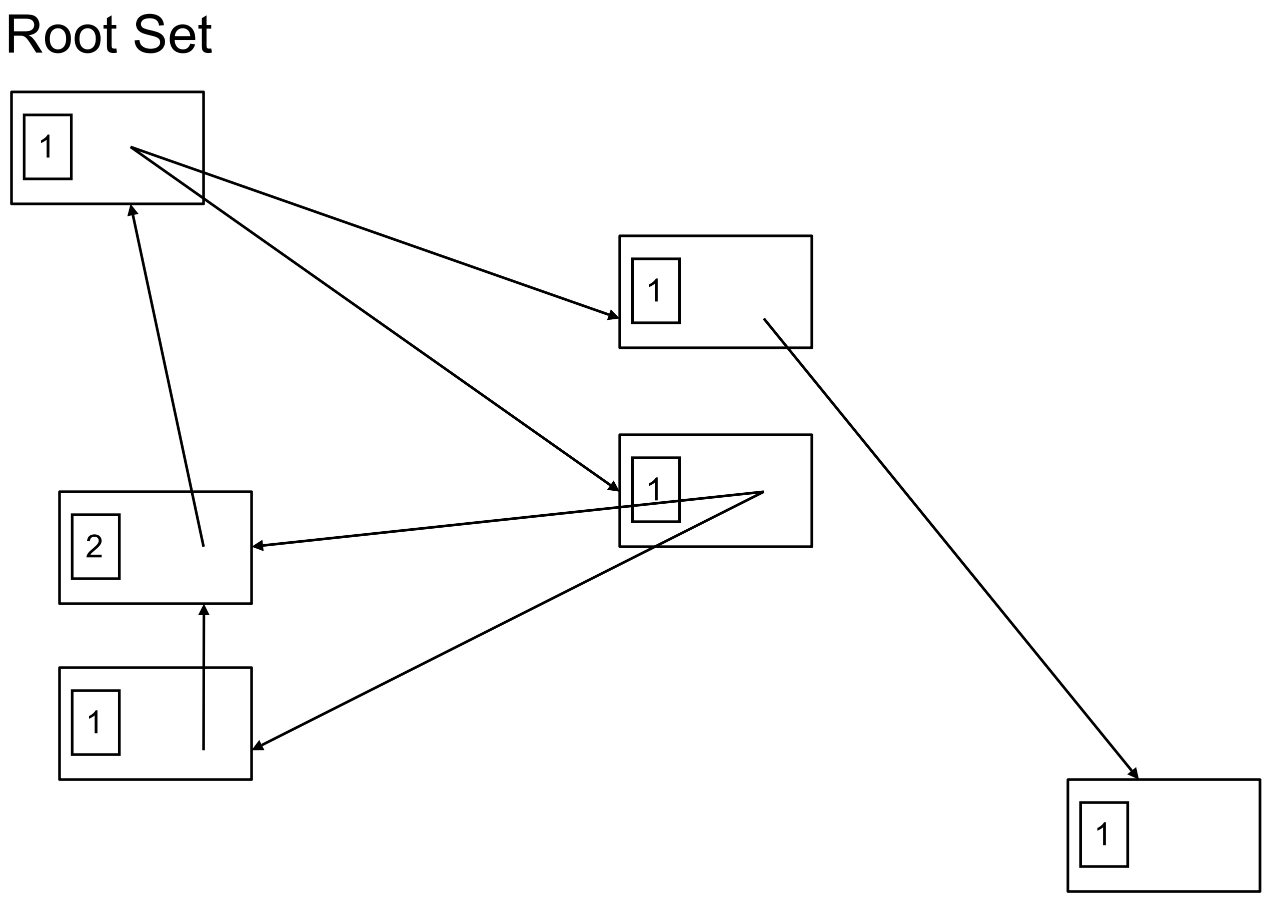
Buddy System	Causes *internal* only	Causes *external* only	Causes *both types*
Slab Allocator	Causes *internal* only	Causes *external* only	Causes *both types*
K&R (Free List Only)	Causes *internal* only	Causes *external* only	Causes *both types*

Garbage Collection

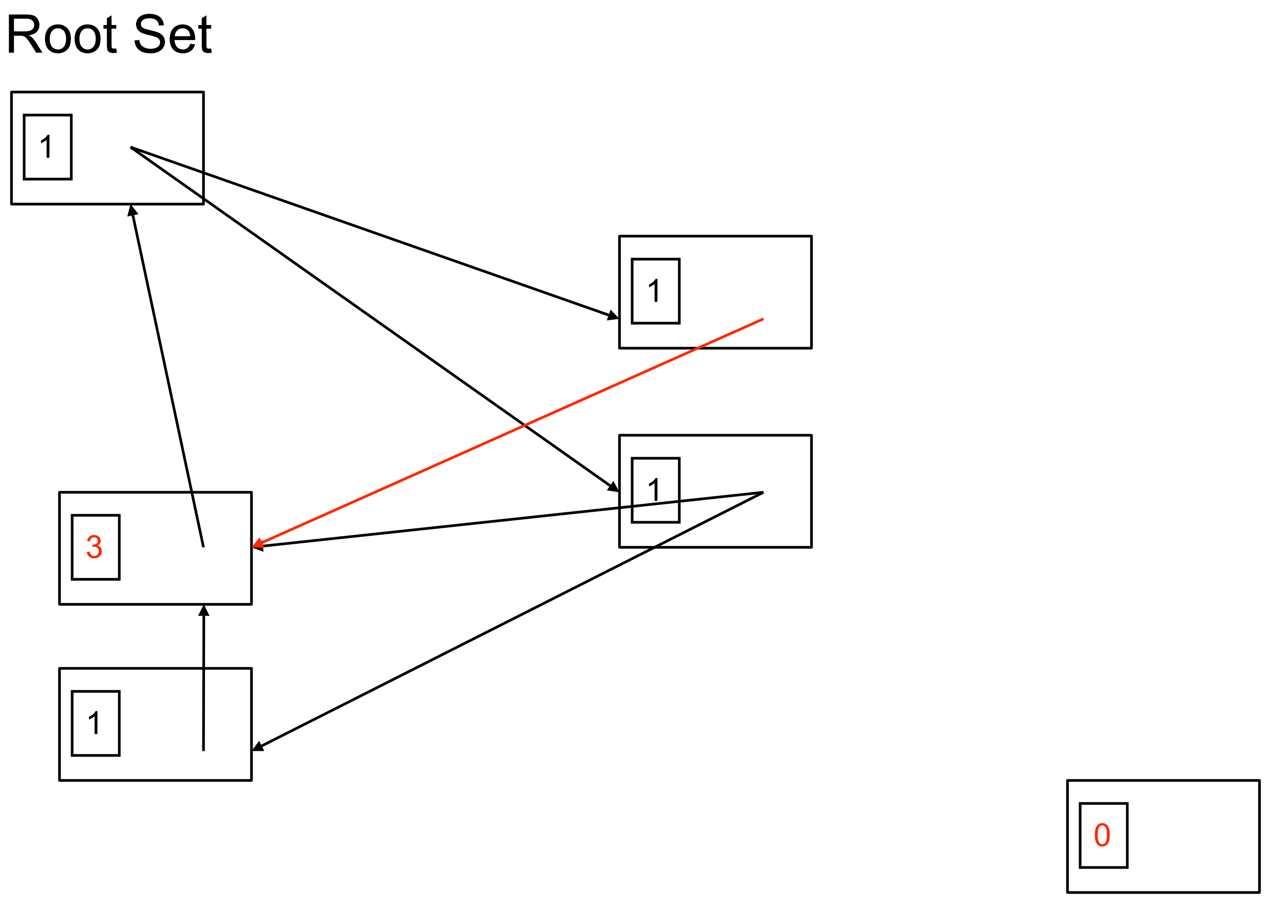
- Garbage collection is used for automatically cleaning up the heap. We can’t do this in C, because of pointer casting, pointer math, etc.

- Lecture covered three different ways to garbage collect:
  - Reference Count
  - Mark and Sweep
  - Stop and Copy

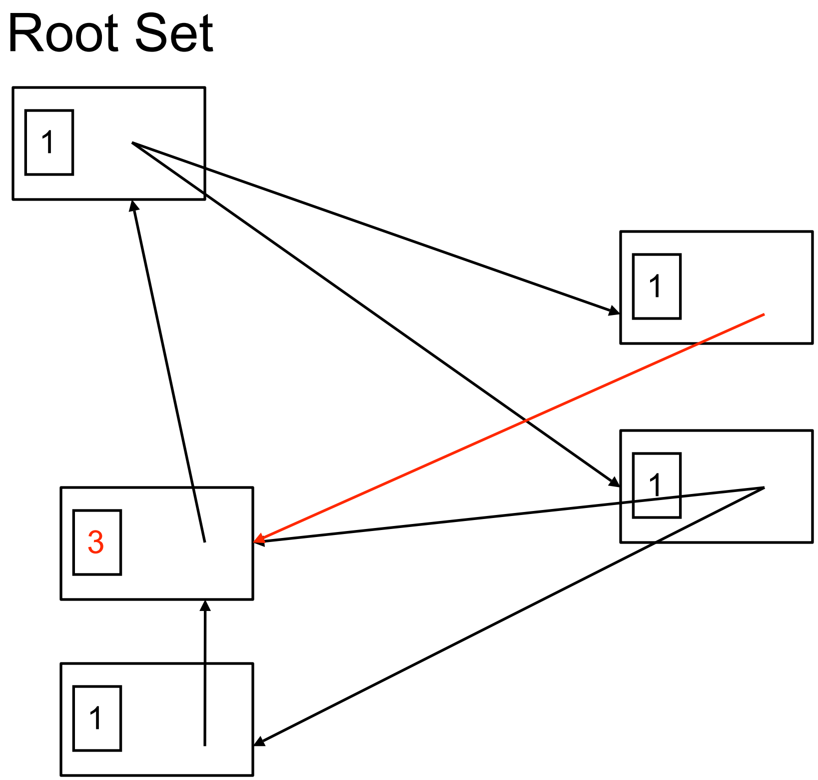
Reference Count

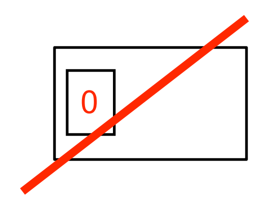


Reference Count



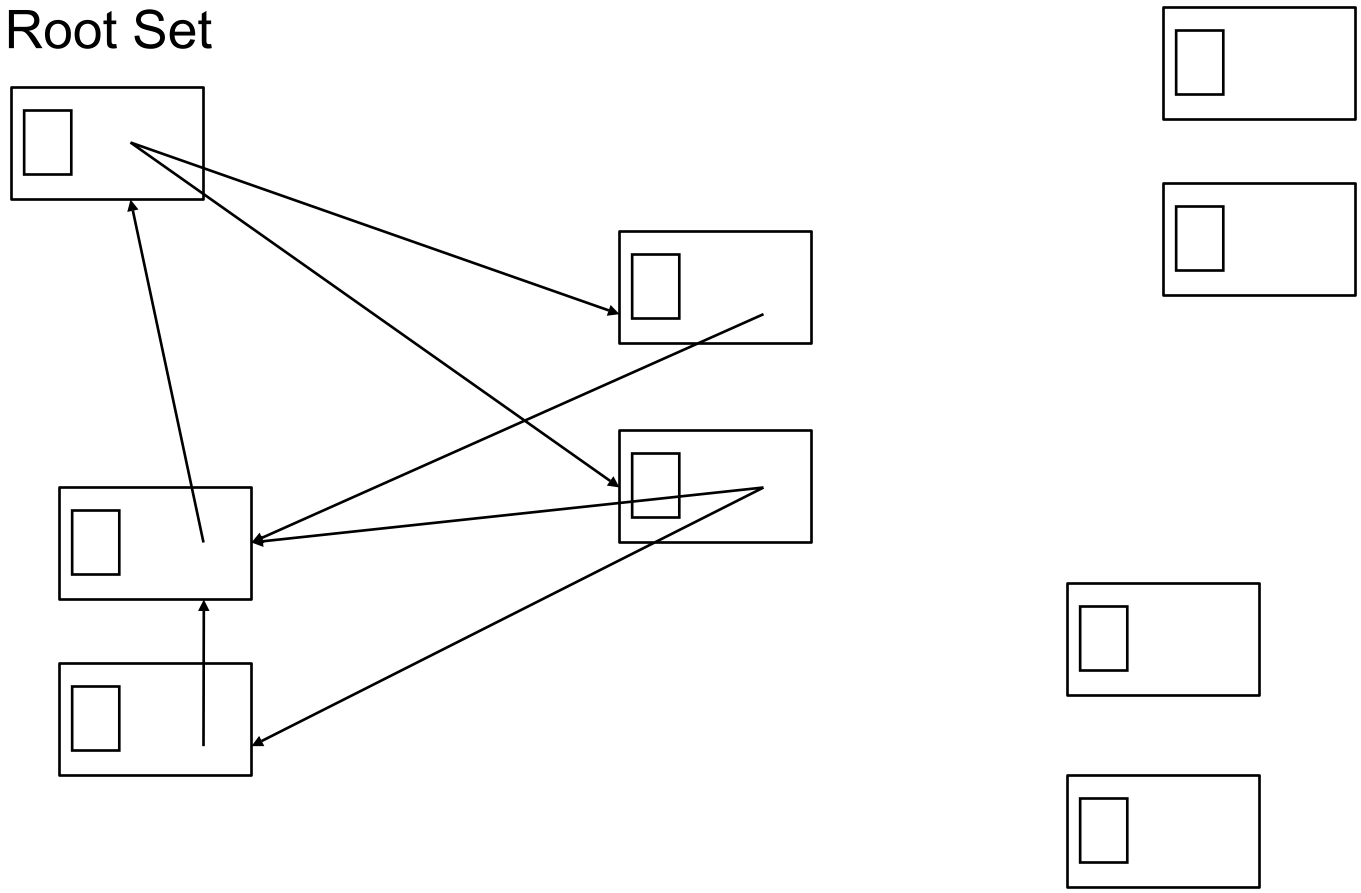
Reference Count



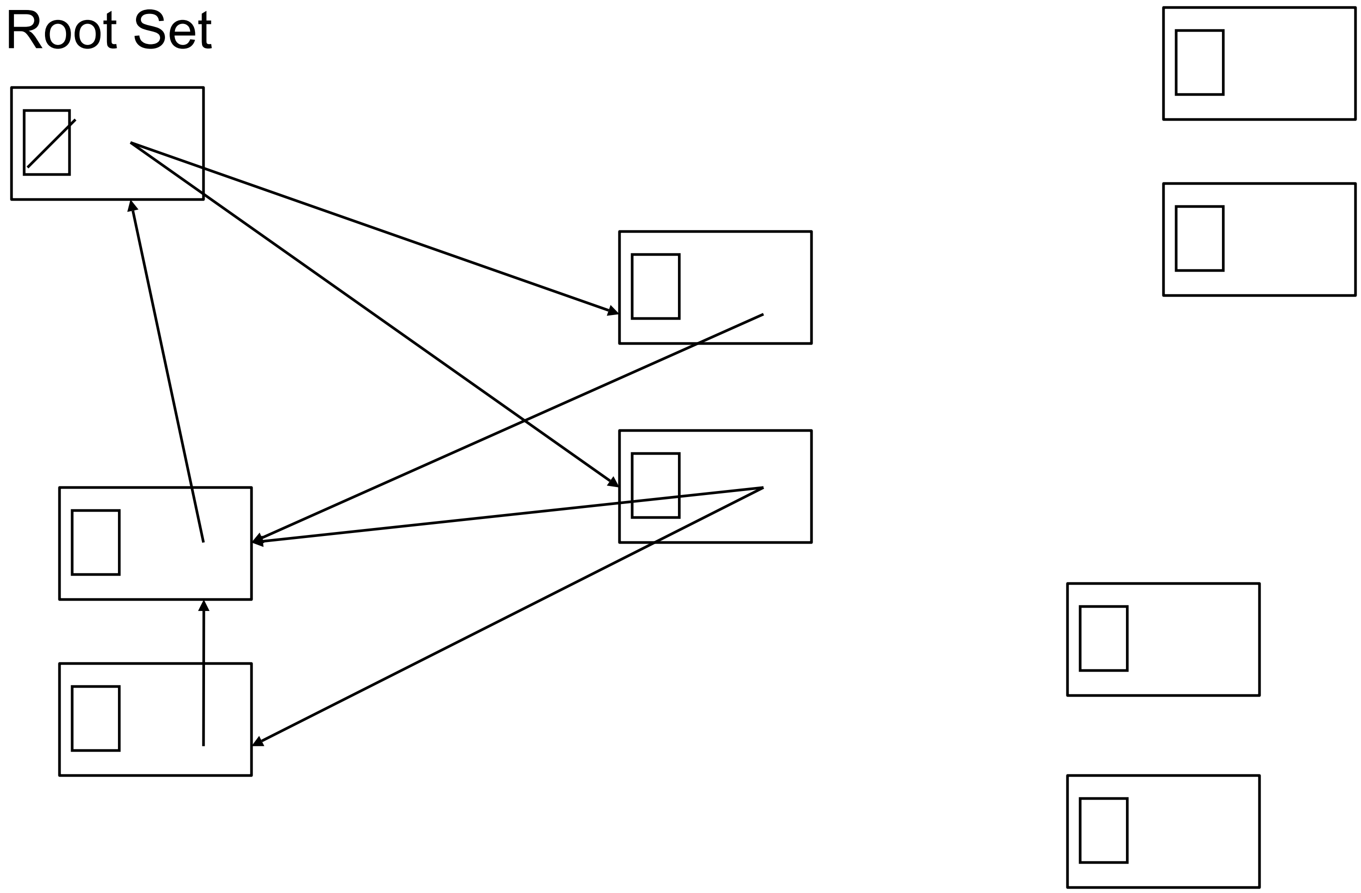


Lots of overhead – every time a pointer changes, the count changes. Unused cycles are never retrieved!

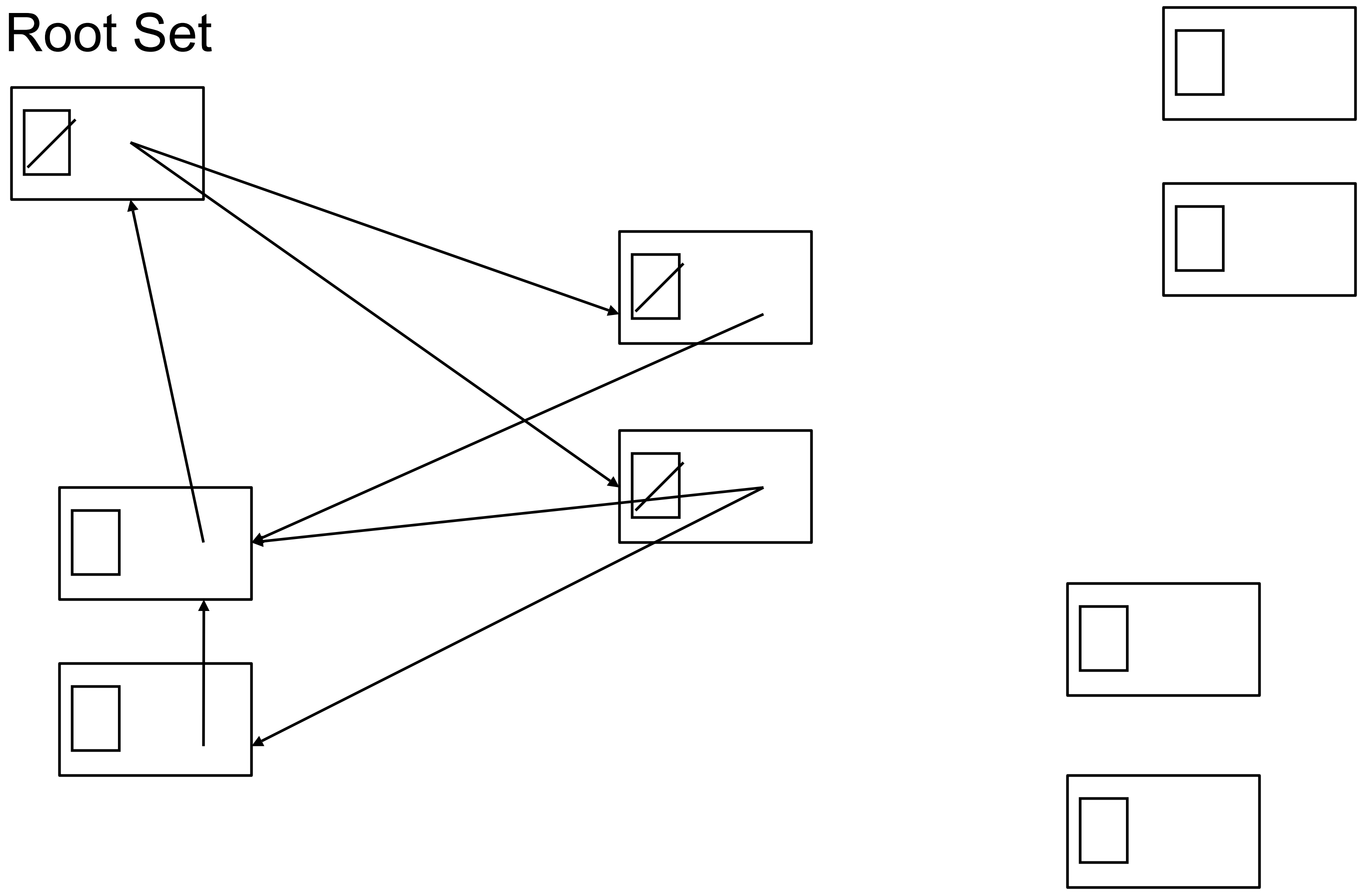
Mark and Sweep



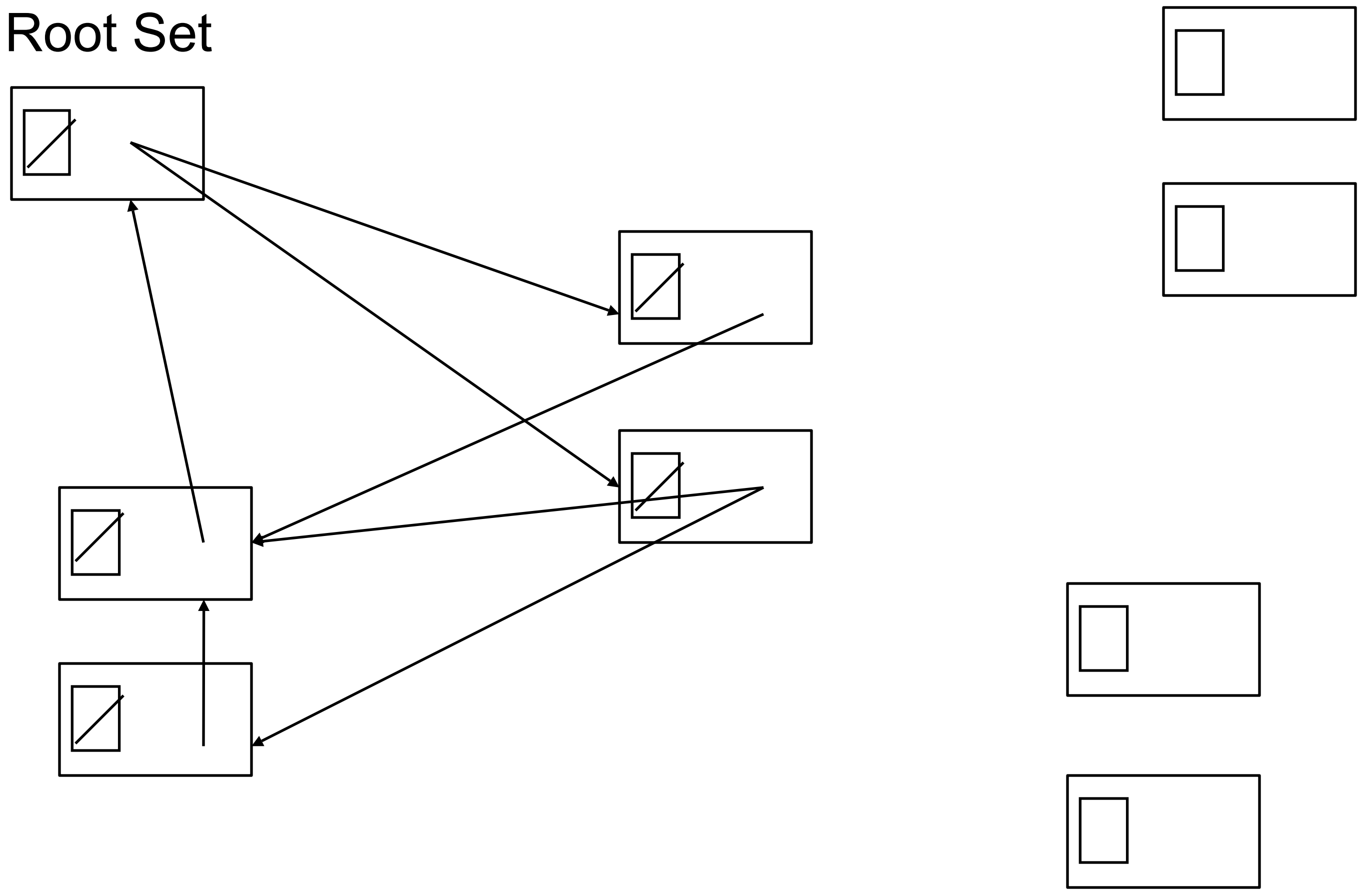
Mark and Sweep



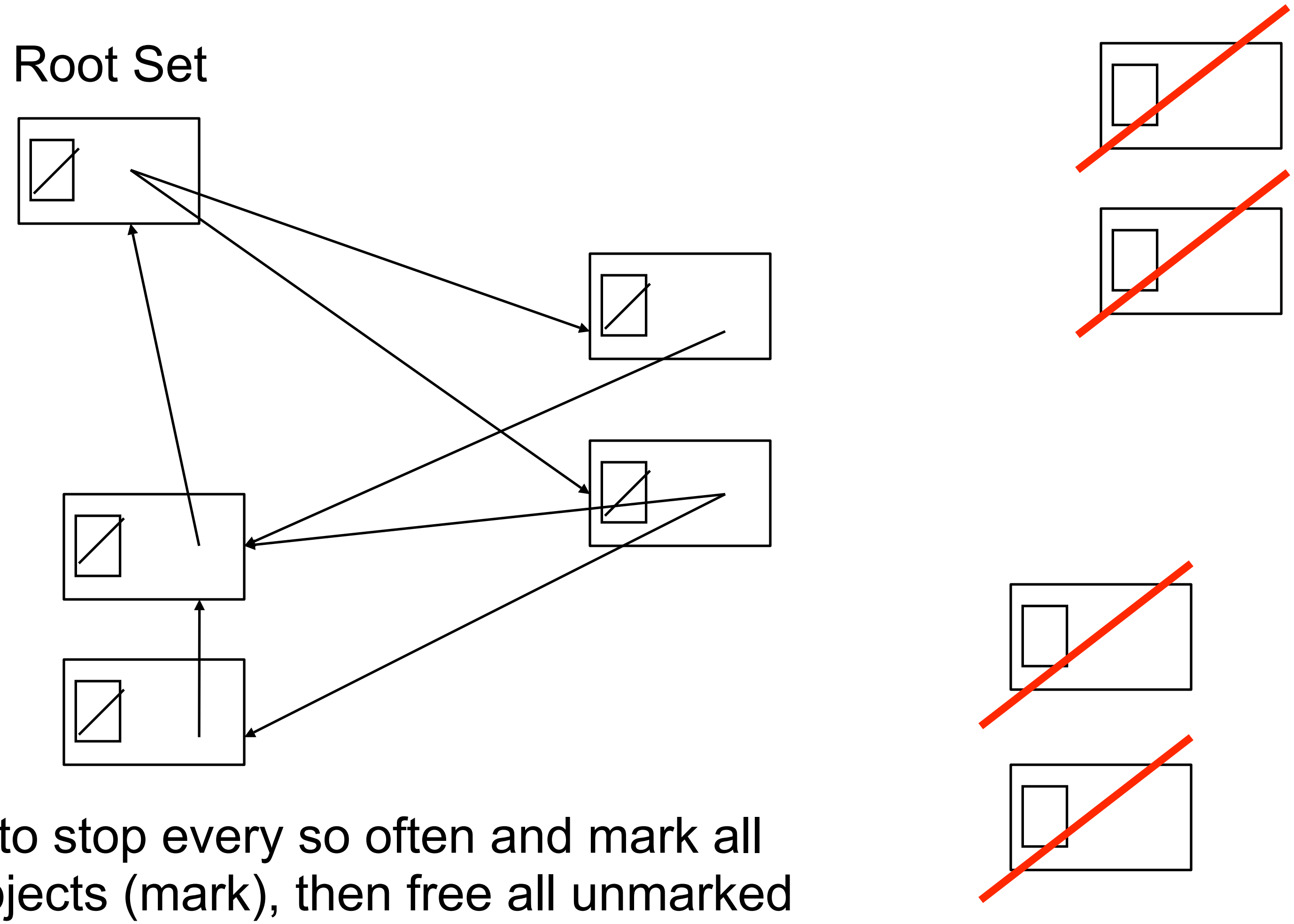
Mark and Sweep



Mark and Sweep

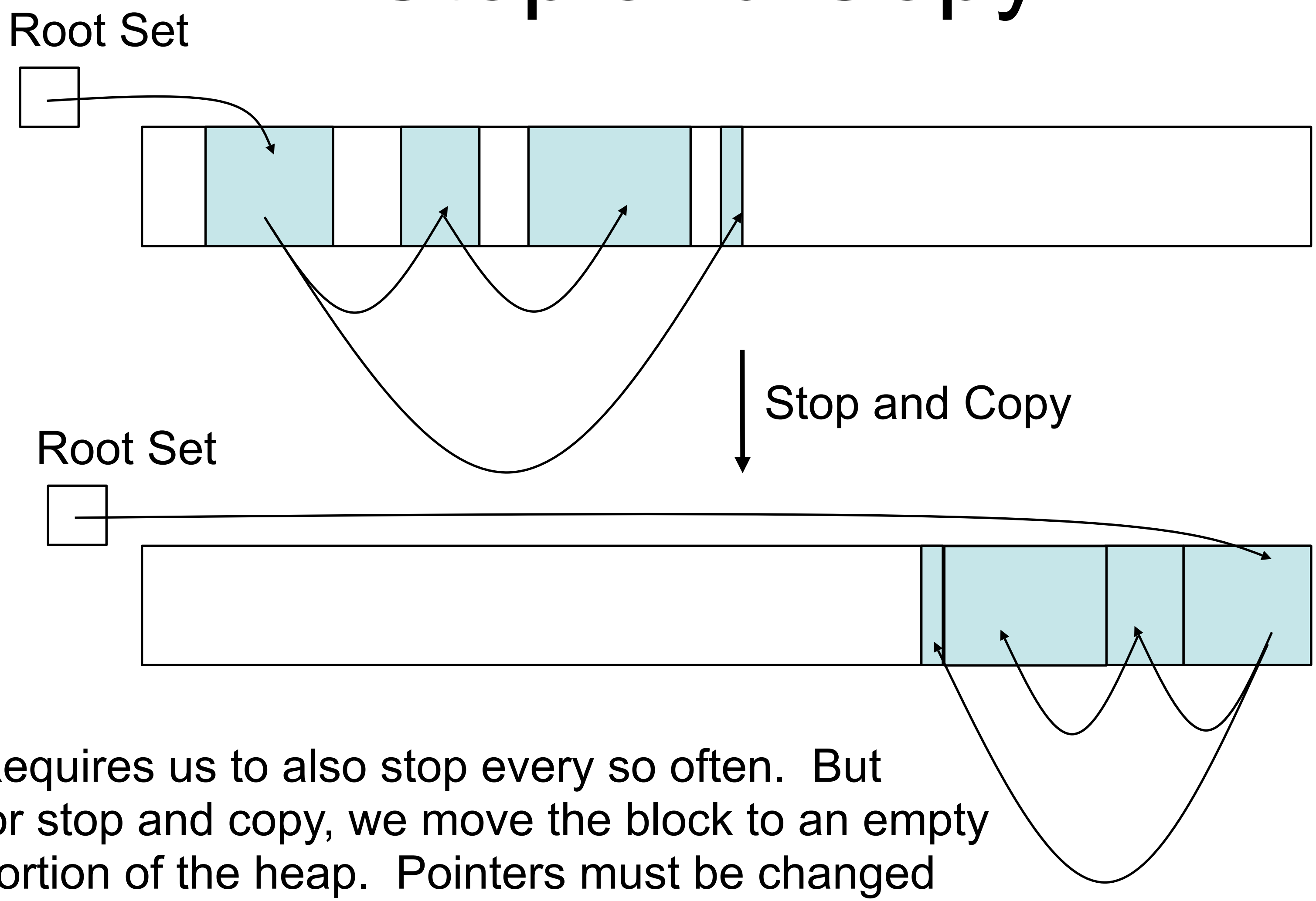


Mark and Sweep



Requires us to stop every so often and mark all reachable objects (mark), then free all unmarked blocks (sweep). Once mark and sweep is done, unmark everything!

Stop and Copy



Requires us to also stop every so often. But for stop and copy, we move the block to an empty portion of the heap. Pointers must be changed to reflect the change in block location. Forwarding pointers must be used!

☺ An Old Midterm Question ☺

- Three code gurus are using different garbage collection techniques on three identical machines (heap memory size *M*). Fill in the table. All answers should be a function of *M*, e.g., *"M/7"* or *"5M"*. (data = data in heap)

What is the…	*most space* their data could take *before* GC	*least* space their data could take *after* GC?	*Most* space their data could take *after* GC	*Most* wasted space that GC can't recover?
Reference Counting				
Mark and Sweep				
Copying				

☺ An Old Midterm Question ☺

- Three code gurus are using different garbage collection techniques on three identical machines (heap memory size *M*). Fill in the table. All answers should be a function of *M*, e.g., *"M/7"* or *"5M"*. (data = data in heap)

What is the…	*most space* their data could take *before* GC	*least* space their data could take *after* GC?	*Most* space their data could take *after* GC	*Most* <u>wasted</u> space that GC can't recover?
Reference Counting	M (-constant)	0	M (-constant)	M (-constant)
Mark and Sweep	M (-constant)	0	M (-constant)	0
Stop & Copy	M/2 (- really small constant)	0	M/2 (- really small constant)	0