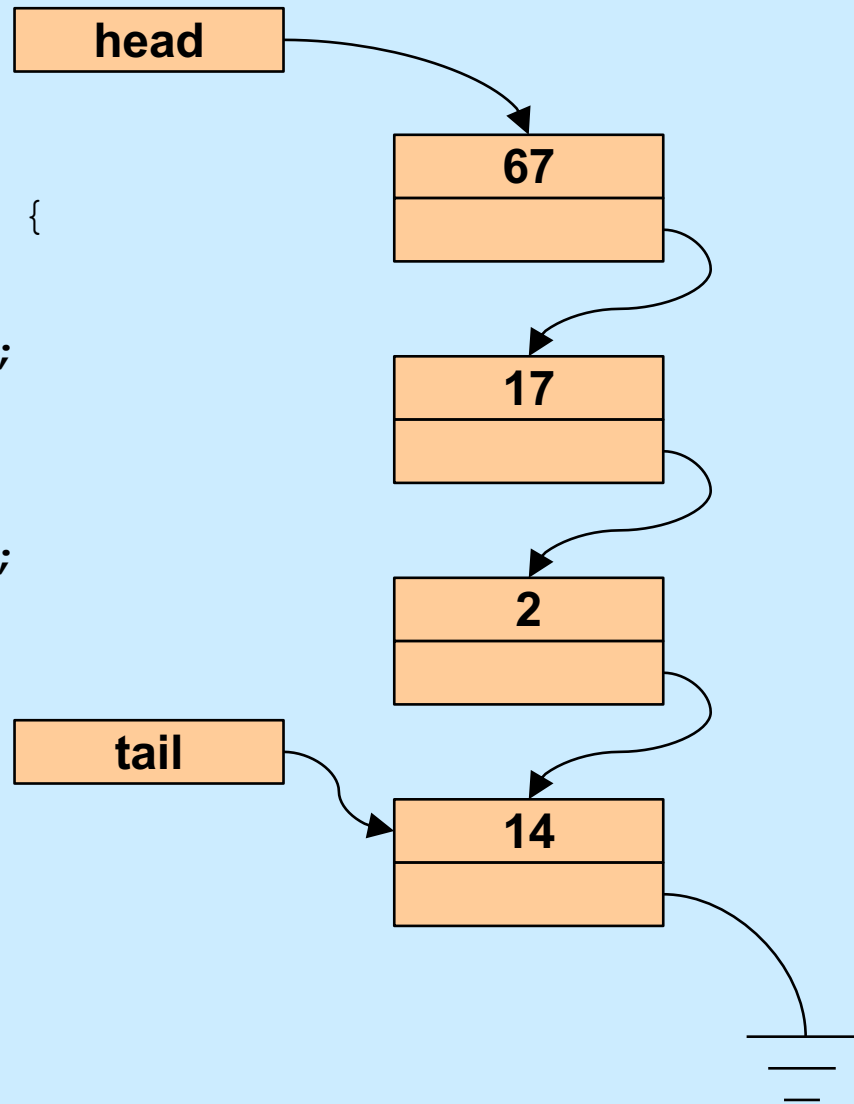


CS 33

Intro to Storage Allocation

A Queue

```
typedef struct list_element {  
    int value;  
    struct list_element *next;  
} list_element_t;  
  
list_element_t *head, *tail;
```



Enqueue

```
int enqueue(int value) {
    list_element_t *newle
        = (list_element_t *)malloc(sizeof(list_element_t));
    if (newle == 0)
        return 0; // can't do it: out of memory
    newle->value = value;
    newle->next = 0;
    if (head == 0) {
        // list was empty
        assert(tail == 0);
        head = newle;
    } else {
        tail->next = newle;
    }
    tail = newle;
    return 1;
}
```

Deque

```
int dequeue(int *value) {
    list_element_t *first;
    if (head == 0) {
        // list is empty
        return 0;
    }
    *value = head->value;
    first = head;
    head = head->next;
    if (tail == first) {
        assert(head == 0);
        tail = 0;
    }
    return 1;
}
```

What's wrong with this code???

Storage Leaks

```
int main() {  
    while(1)  
        if (malloc(sizeof(list_element_t)) == 0)  
            break;  
    return 1;  
}
```

**For how long will this program
run before terminating?**

Dequeue, Fixed

```
int dequeue(int *value) {
    list_element_t *first;
    if (head == 0) {
        // list is empty
        return 0;
    }
    *value = head->value;
    first = head;
    head = head->next;
    if (tail == first)
        assert(head == 0);
    tail = 0;
}
free(first);
return 1;
}
```

Quiz 1

```
int enqueue(int value) {
    list_element_t *newle
        = (list_element_t *)malloc(sizeof(list_element_t));
    if (newle == 0)
        return 0;
    newle->value = value;
    newle->next = 0;
    if (head == 0) {
        // list was empty
        assert(tail == 0);
        head = newle;
    } else {
        tail->next = newle;
    }
    tail = newle;
    free(newle); // saves us the bother of freeing it later
    return 1;
}
```

This version of enqueue makes unnecessary the call to free in dequeue.

- a) It never works.**
- b) It hardly ever works.**
- c) It fails occasionally.**
- d) It works well.**

malloc and free

```
void *malloc(size_t size)
```

- allocate *size* bytes of storage and return a pointer to it
- returns 0 (NULL) if the requested storage isn't available

```
void free(void *ptr)
```

- free the storage pointed to by *ptr*
- *ptr* must have previously been returned by *malloc* (or other storage-allocation functions — *calloc* and *realloc*)



realloc

```
void *realloc(void *ptr, size_t size)
```

- change the size of the storage pointed to by *ptr*
- the contents, up to the minimum of the old size and new size, will not be changed
- *ptr* must have been returned by a previous call to *malloc*, *realloc*, or *calloc*
- it may be necessary to allocate a completely new area and copy from the old to the new
 - » thus the return value may be different from *ptr*
 - » if copying is done the old area is freed
- returns 0 if the operation cannot be done

Get (contiguous) Input (1)

```
char *getinput() {
    int alloc_size = 4; // start small
    int read_size = 4; // max number of bytes to read
    int next_read = 0; // index in buf of next read
    int bytes_read; // number of bytes read
    char *buf = (char *)malloc(alloc_size);
    char *newbuf;

    if (buf == 0) {
        // no memory
        return 0;
    }
}
```

Get (contiguous) Input (2)

```
while (1) {  
    if ((bytes_read  
        = read(0, buf+next_read, read_size)) == -1) {  
        perror("getinput");  
        return 0;  
    }  
    if (bytes_read == 0) {  
        // eof  
        break;  
    }  
    if ((buf+next_read)[bytes_read-1] == '\n') {  
        // end of line  
        break;  
    }  
}
```

Get (contiguous) Input (3)

```
next_read += read_size;
read_size = alloc_size;
alloc_size *= 2;
newbuf = (char *)realloc(buf, alloc_size);
if (newbuf == 0) {
    // realloc failed: not enough memory.
    // Free the storage allocated previously and report
    // failure.
    free(buf);
    return 0;
}
buf = newbuf;
}
```

Get (contiguous) Input (4)

```
// reduce buffer size to the minimum necessary
newbuf = (char *)realloc(buf,
    alloc_size - (read_size - bytes_read));
if (newbuf == 0) {
    // couldn't allocate smaller buf
    return buf;
}
return newbuf;
}
```

Some Common Memory-Related Errors

Dereferencing Bad Pointers

- The classic `scanf` bug

```
int val;  
  
...  
  
scanf("%d", val);
```

Reading Uninitialized Memory

- Assuming that dynamically allocated data is initialized to zero

```
/* return y = Ax */
int *matvec(int A[][N], int x[]) {
    int *y = (int *)malloc(N*sizeof(int));
    int i, j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] += A[i][j]*x[j];
    return y;
}
```

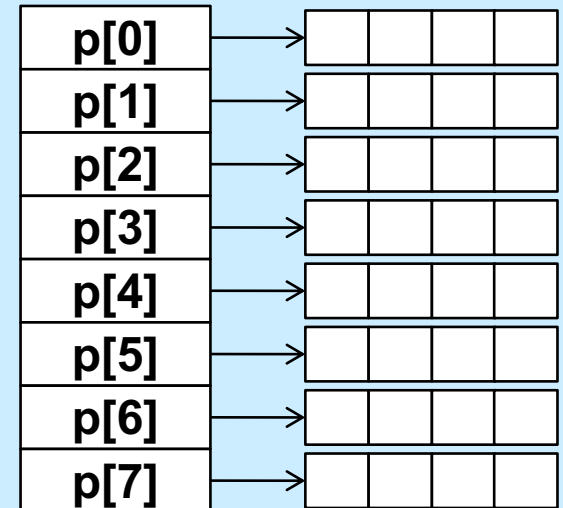

Overwriting Memory

- Allocating the (possibly) wrong-sized object

```
// set up p so it is an array of
// int *'s, allocated dynamically
int **p;

p = (int **)malloc(N*sizeof(int));

for (i=0; i<N; i++) {
    p[i] = (int *)malloc(M*sizeof(int));
}
```



Overwriting Memory

- Not checking the max string size

```
char s[8];  
int i;  
  
gets(s); /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks

Going Too Far

- **Misunderstanding pointer arithmetic**

```
int *search(int p[], int val) {  
  
    while (*p && *p != val)  
        p += sizeof(int);  
  
    return p;  
}
```

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```
int *foo () {  
    int val;  
  
    return &val;  
}
```

Freeing Blocks Multiple Times

```
x = (int *)malloc(N*sizeof(int));  
    <manipulate x>  
free(x);  
  
y = (int *)malloc(M*sizeof(int));  
    <manipulate y>  
free(x);
```

Referencing Freed Blocks

```
x = (int *)malloc(N*sizeof(int));  
  <manipulate x>  
free(x);  
  ...  
y = (int *)malloc(M*sizeof(int));  
for (i=0; i<M; i++)  
  y[i] = x[i]++;
```

Failing to Free Blocks (Memory Leaks)

```
foo() {  
    int *x = (int *)malloc(N*sizeof(int));  
    Use(x, N);  
    return;  
}
```

Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```
struct list {
    int val;
    struct list *next;
};

foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    <allocate and manipulate the rest of the list>
    ...
    free(head);
    return;
}
```


Total Confusion

```
foo() {  
    char *str;  
    str = (char *)malloc(1024);  
    ...  
    str = "";  
    ...  
    strcat(str, "c");  
    ...  
    return;  
}
```

It Works, But ...

- Using a hammer where a feather would do ...

```
hammer() {  
    int *x = (int *)malloc(1024*sizeof(int));  
    Use(x, 1024);  
    free(x);  
    return;  
}
```

```
feather() {  
    int x[1024];  
    Use(x, 1024);  
    return;  
}
```

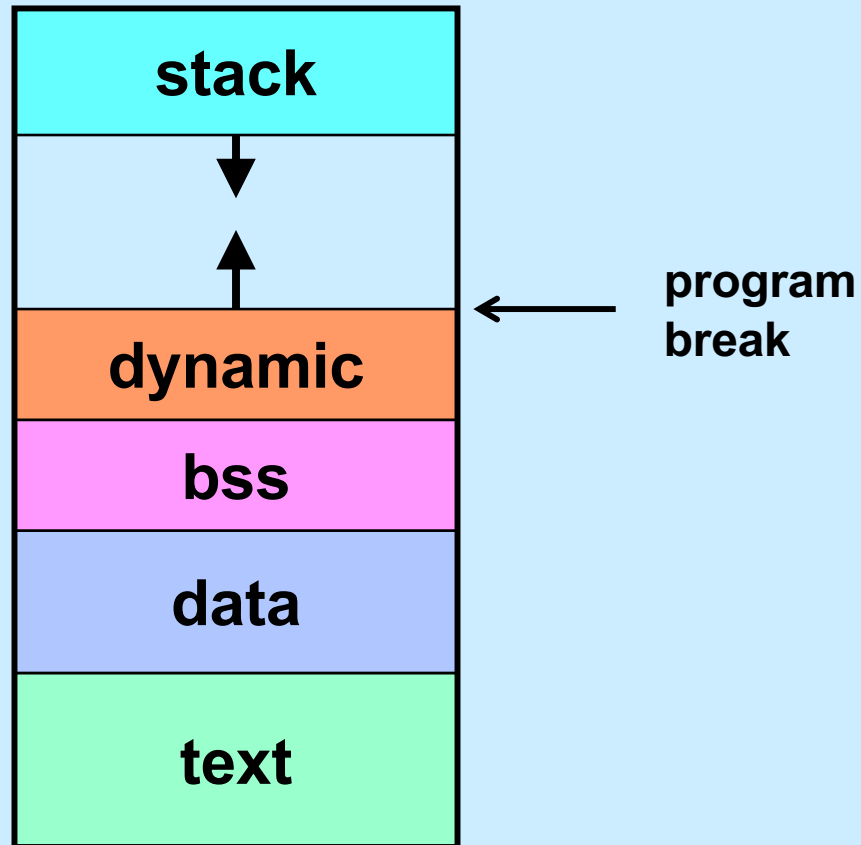
Quiz 2

- Will this work?
 - a) never
 - b) usually
 - c) always

```
typedef struct  
TwoParts {  
    int part1[120];  
    float part2[200];  
} TwoParts_t;
```

```
void func() {  
    TwoParts_t *X;  
    X = malloc(sizeof(TwoParts_t));  
    UseX1(X->part1);  
    free(&X->part1);  
    UseX2(X->part2);  
    free(&X->part2);  
}
```

The Unix Address Space



sbrk System Call

```
void *sbrk(intptr_t increment)
```

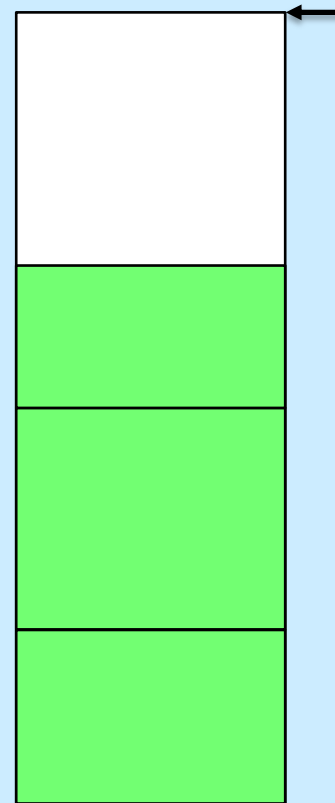
- moves the program break by an amount equal to *increment*
- returns the previous program break
- *intptr_t* is typedef'd to be a *long*

Managing Dynamic Storage

- **Strategy**
 - get a “chunk” of memory from the OS using *sbrk*
 - » create pool of available storage, aka the “heap”
 - *malloc*, *calloc*, *realloc*, and *free* use this storage if possible
 - » they manage the heap
 - if not possible, get more storage from OS
 - » heap is made larger (by calling *sbrk*)
- **Important note:**
 - when process terminates, all storage is given back to the system
 - » all memory-related sins are forgotten!

Malloc and Free

```
x = malloc(40);  
y = malloc(60);  
z = malloc(30);  
free(y);
```

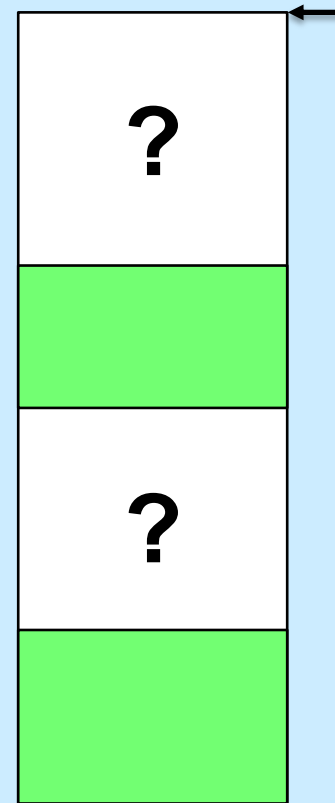


Malloc and Free

```
x = malloc(40);  
y = malloc(60);  
z = malloc(30);  
free(y);
```

```
w = malloc(60);
```

- How do we keep track of where free space is?
- How do we choose which to use?



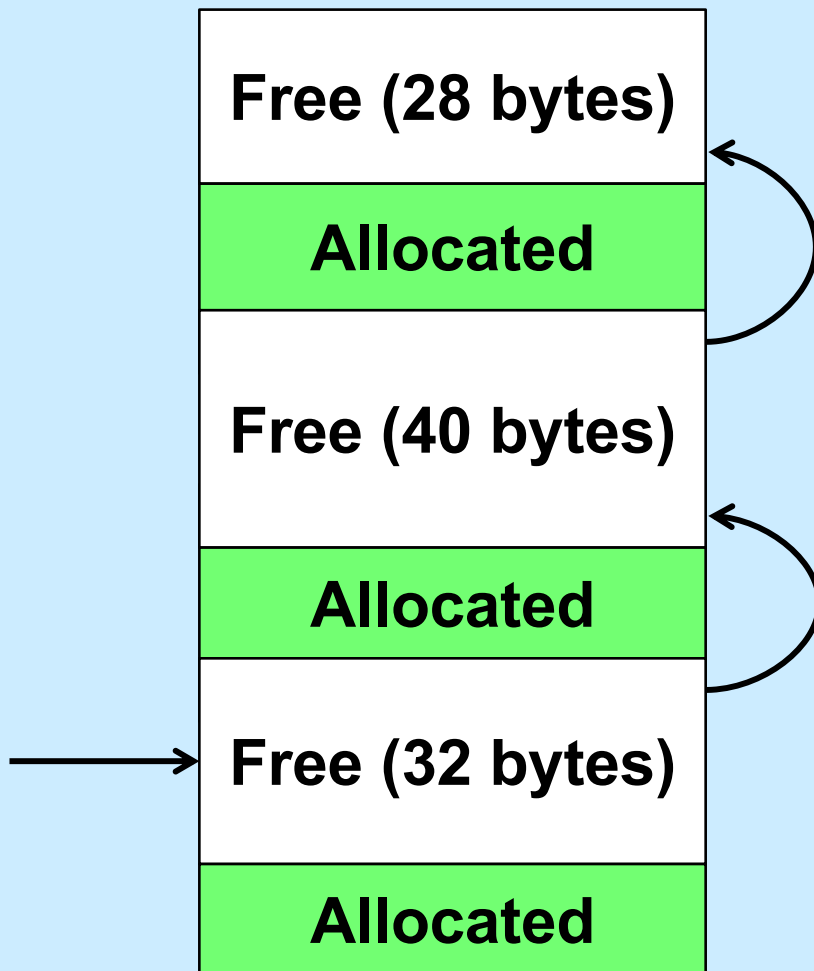
Managing Free Space

- **Two possibilities**
 - 1) **don't worry about it: memory is cheap and plentiful — simply call *sbrk* when a new block is needed**
 - 2) **link together the free blocks**

Finding the Right Free Block

`malloc(24)`

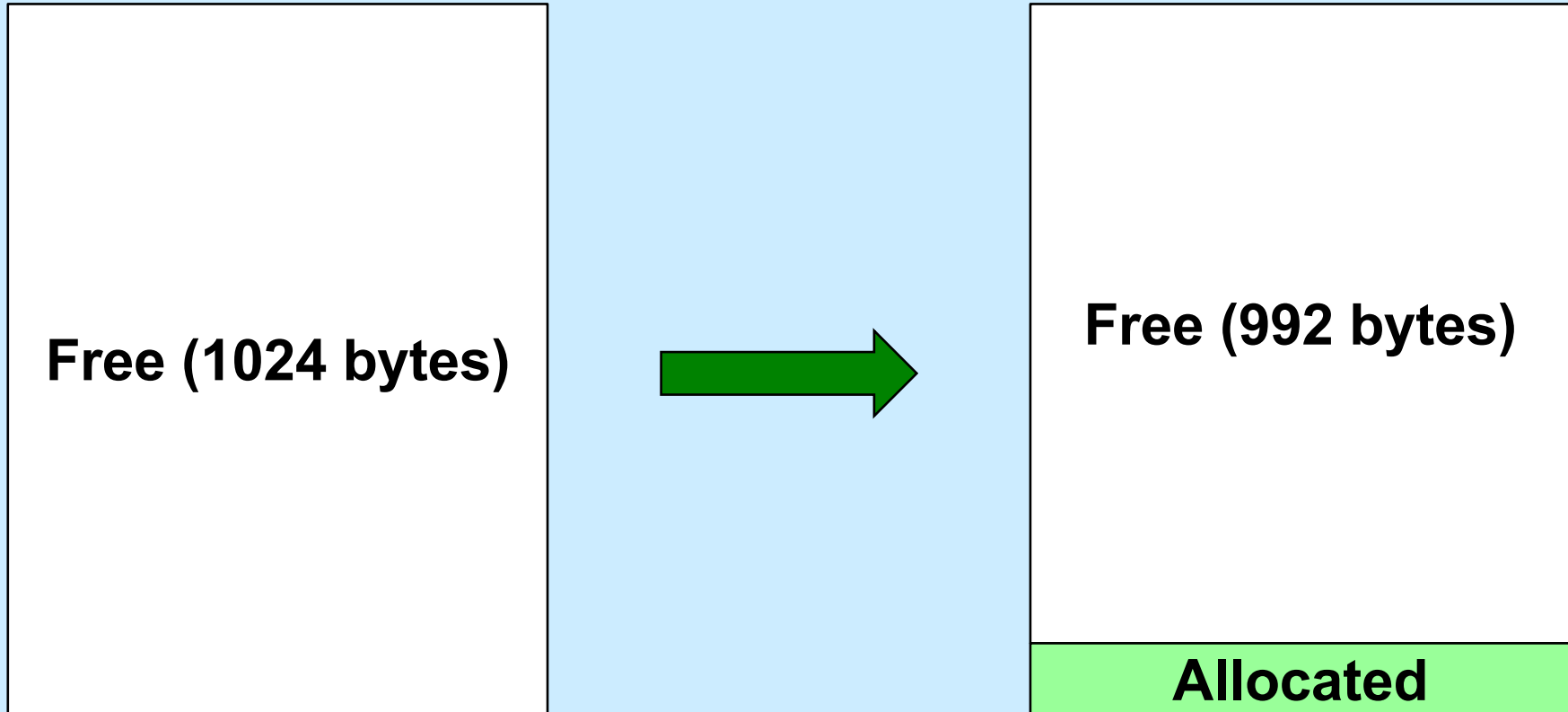
- Search strategies
 - first fit
 - best fit



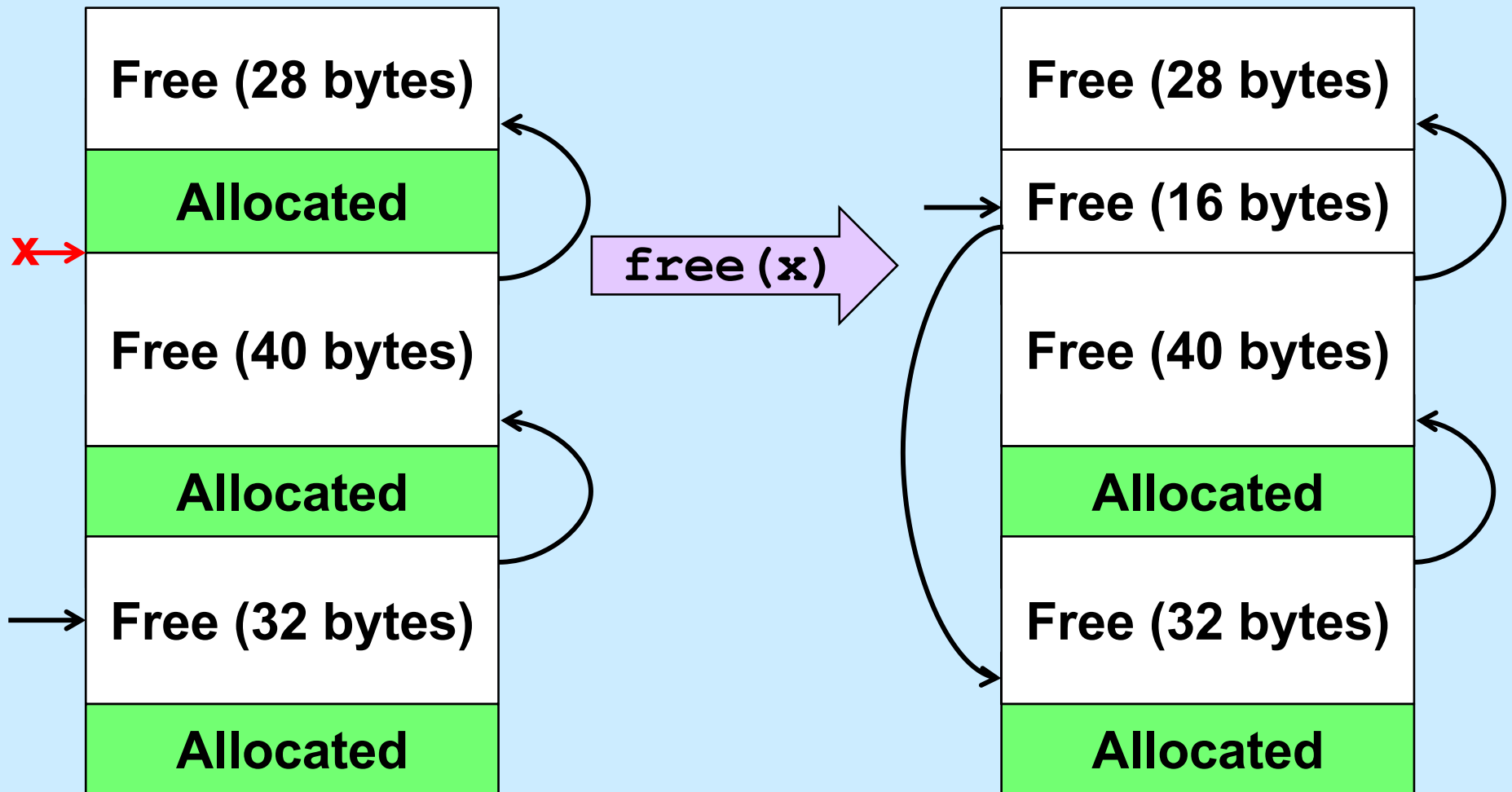
A Problem

- A malloc request is for a block of 32 bytes
- The block found on the free list is 1024 bytes long
- Should malloc return a pointer to the entire 1024-byte block?

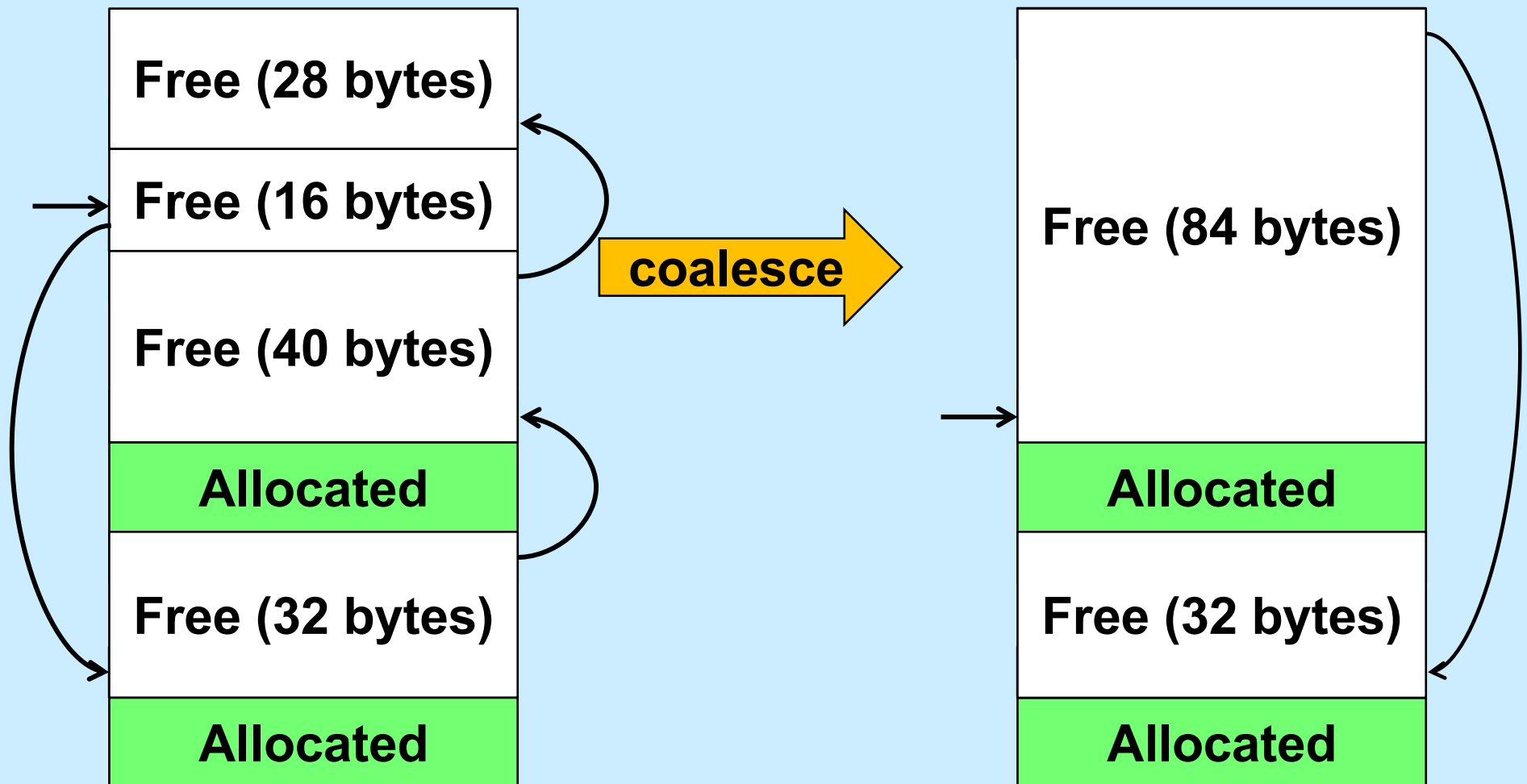
Splitting



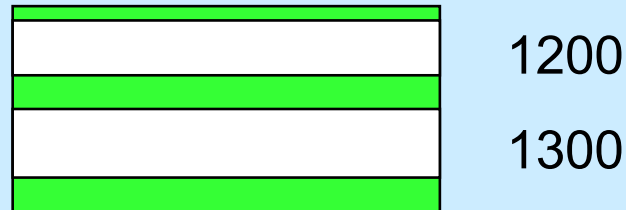
Another Problem



Coalescing



Quiz 3



We have two free blocks of memory, of sizes 1300 and 1200 (appearing in that order). There are three successive requests to *malloc* for allocations of 1000, 1100, and 250 bytes. Which approach does best? (Hint: one of the two fails the last request.)

- a) best fit**
- b) first fit**

Allocation



1200

1300

First Fit



1200

1000 bytes

300



100

1100 bytes

300



100

250 bytes

50

Best Fit



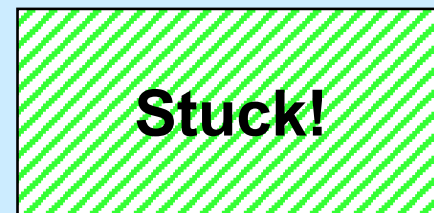
200

1300



200

200



Stuck!

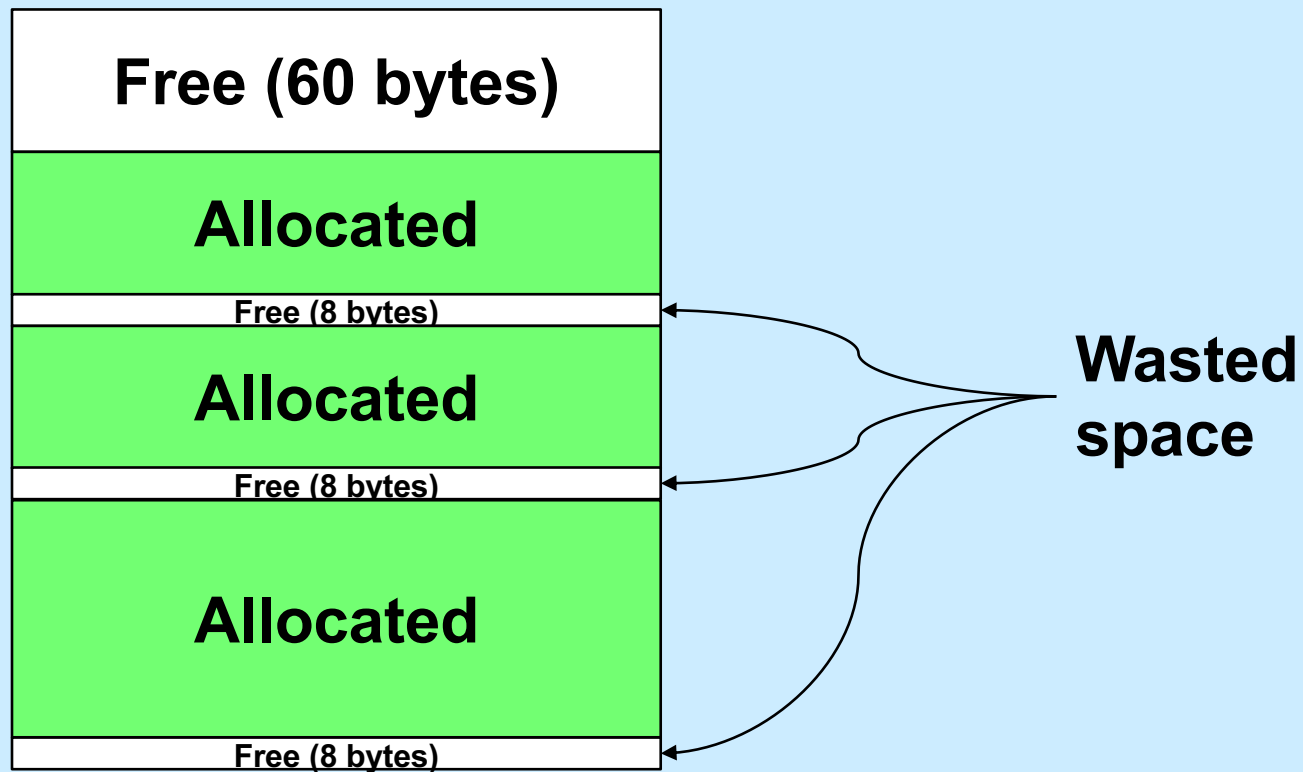
Some Observations

- **Best fit**
 - perhaps leaves behind chunks that are too small to be of use
 - requires linear time (in size of free list) for malloc
- **First fit**
 - small chunks congregate at beginning of free list
 - upper bound of linear time for malloc, but often much less

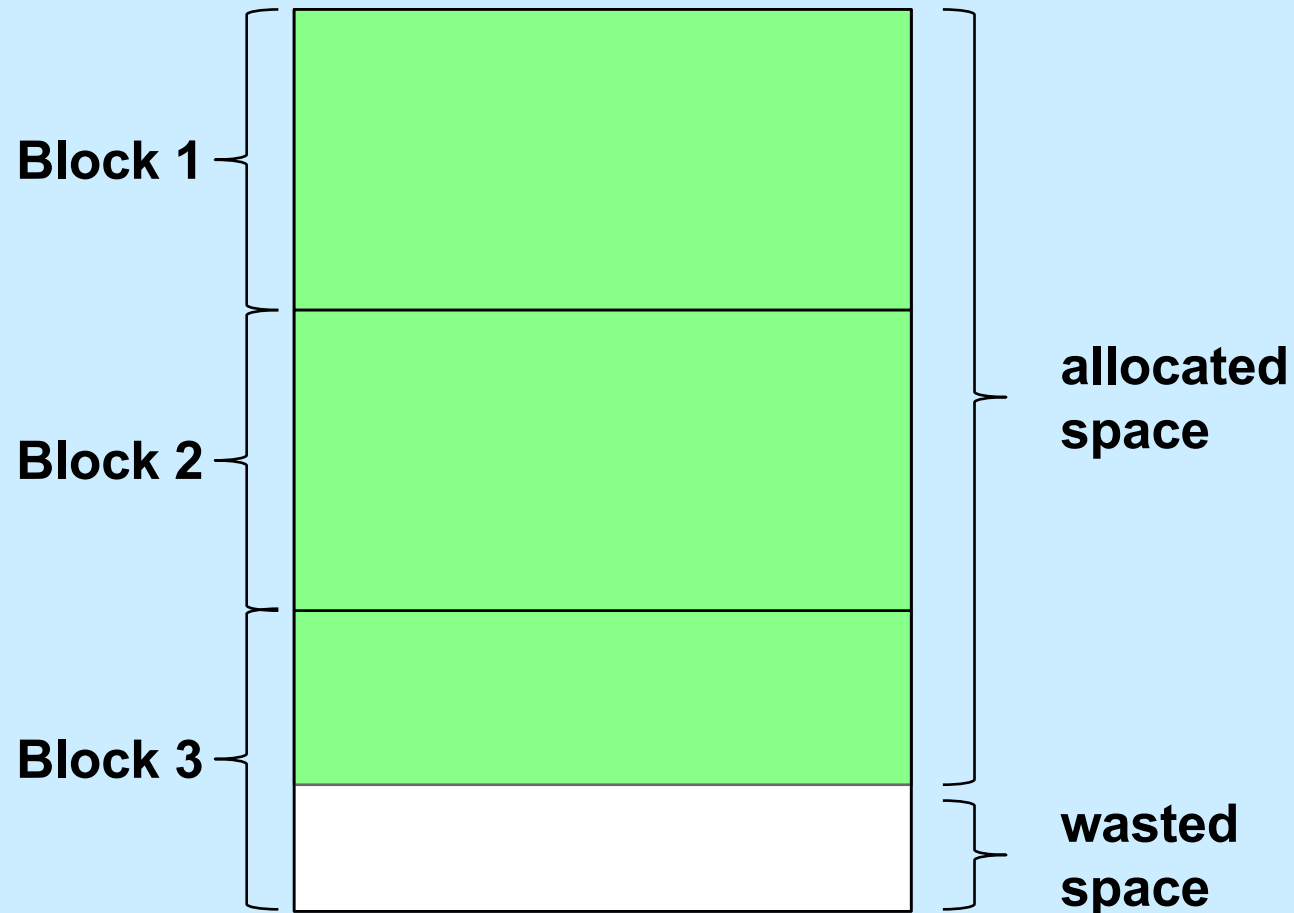
Fragmentation

- **Fragmentation refers to the wastage of memory due to our allocation policy**
- **Two sorts**
 - external fragmentation
 - internal fragmentation

External Fragmentation



Internal Fragmentation



Variations

- **Next fit**
 - like first fit, but the next search starts where the previous ended
- **Worst fit**
 - always allocate from largest free block
 - » perhaps reduces the number of “too small” blocks
- **Free-list insertion**
 - LIFO
 - » easy to do
 - » $O(1)$
 - ordered insertion
 - » $O(n)$

Quiz 4

Assume that best-fit results in less external fragmentation than first-fit.

We are running an application with modest memory demands. Which allocation strategy is likely to result in better performance (in terms of time) for the application:

- a) best-fit**
- b) first-fit with ordered insertion**
- c) first-fit with LIFO insertion**