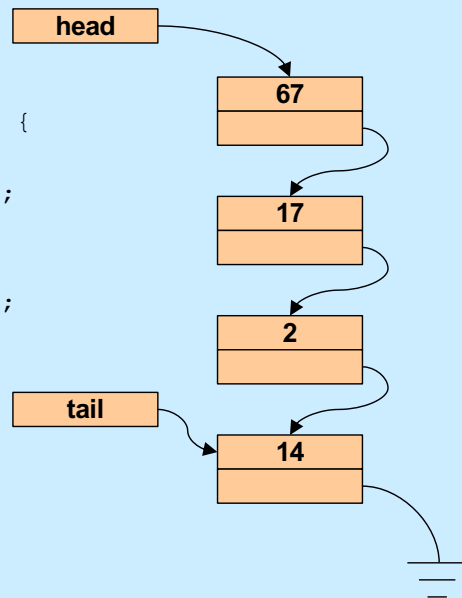


# CS 33

## 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;
}
```

Note that **malloc** allocates storage to hold a new instance of **list\_element\_t**.

## 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???**

The problem with this code, which removes the first item in the queue, is that the list element being removed is lost – its storage is not returned to the pool of free memory.

## 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?**

Answer: around 3 minutes on a SunLab machine.

## 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;
}
```

Here after removing the list element from the list, we return it to the pool of free memory by calling *free*.

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

## 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*)



When something is malloc'd, the system must keep track of its size. Thus, when it's freed, the system will know how much storage is being freed.



## 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;
    }
}
```

In this example, we're to read a line of input, where a line is delineated by a newline character. However, we have no upper bound on its length. So, we start by allocating four bytes of storage for the line. If that's not enough (the four bytes read in don't end with a '\n'), we then double our allocation and read in more up to the end of the new allocation, if that's not enough, we double the allocation again, and so forth. When we're finished, we reduce the allocation, giving back to the system that portion we didn't need.

## 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;
    }
}
```

We assume that if read returns neither -1 nor 0, then either it has filled the buffer or that the last character read in was '\n'.

## 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;
}
```

If we get here, then it's the case that the buffer wasn't big enough. So, let's try to get a larger buffer. If we can't get a larger buffer (e.g., the system is out of memory), we free up everything and report failure (probably not a great way to handle this, but it's convenient for the slide).

## 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);
```

Supplied by CMU.

## 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;  
}
```

Supplied by CMU.

This code multiplies an  $N \times N$  matrix and a vector of length  $N$ , returning a pointer to a (newly allocated) vector of length  $N$ .



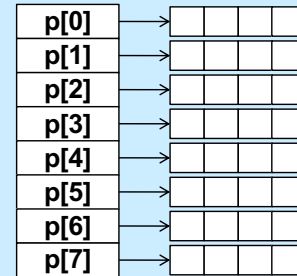
# Overwriting Memory

- 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));
}
```



Supplied by CMU.

The problem here is that the storage allocated for `p` is of size `N*sizeof(int)`, when it should be `N*sizeof(int *)` — on a 64-bit machine, `p` won't have been assigned enough storage.

# 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

Supplied by CMU.

## Going Too Far

- Misunderstanding pointer arithmetic

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

Supplied by CMU.

## Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

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

Supplied by CMU.

## Freeing Blocks Multiple Times

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

Supplied by CMU.

## 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]++;
```

Supplied by CMU.

## Failing to Free Blocks (Memory Leaks)

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

Supplied by CMU.

## 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;  
}
```

Supplied by CMU.



# Total Confusion

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

There are two problems here: space is allocated for **str** to point to, but the space is not freed when **str** no longer points to it. **str** now points to the string "", a string consisting of just the null byte that's in read-only storage. The **strcat** attempts to copy a string into the storage, but not only is the string to be copied too long, but there will be a segfault when the attempt is made to copy it into the read-only storage.

## 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;  
}
```

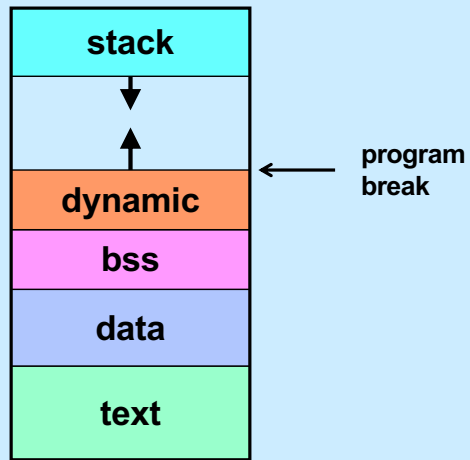
## Quiz 2

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

```
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



The program break is the upper limit of the currently allocated dynamic region.

## 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*

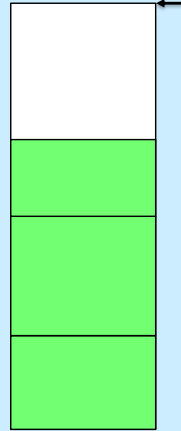
The **increment** is signed, and thus may be positive or negative (if it's zero, the current breakpoint is returned and nothing else happens). If the increment is positive, we're increasing the size of the dynamic area. If it's negative, we're decreasing its size.

# 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);
```



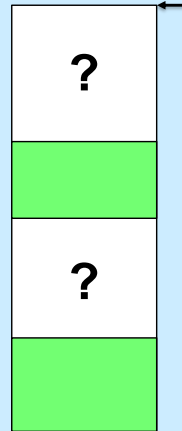
The arrow points to the current program break, indicating the end of the dynamic region. We want to use **malloc** and **free** to manage the memory in the current dynamic region.

# 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?



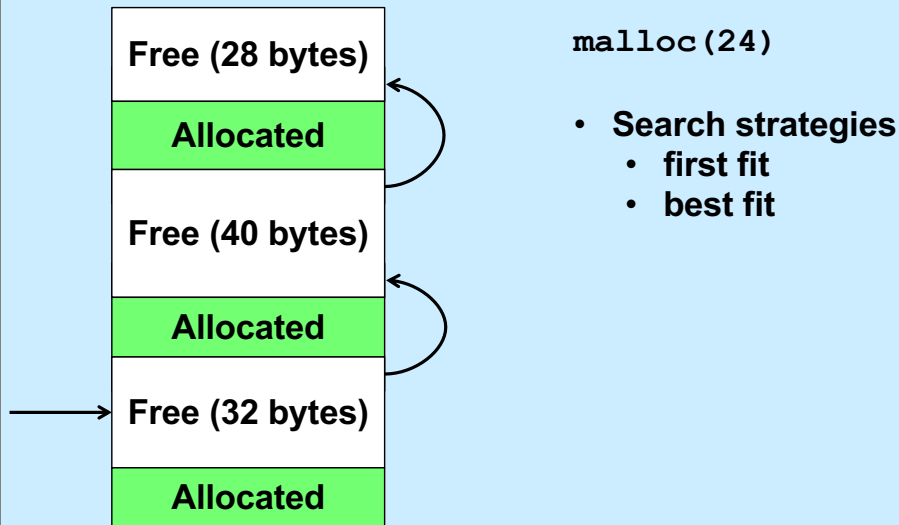
Somehow we need to keep track of where the free space is, so we can use it to handle allocation requests. The green areas of memory are allocated, the white areas are unallocated.



# 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

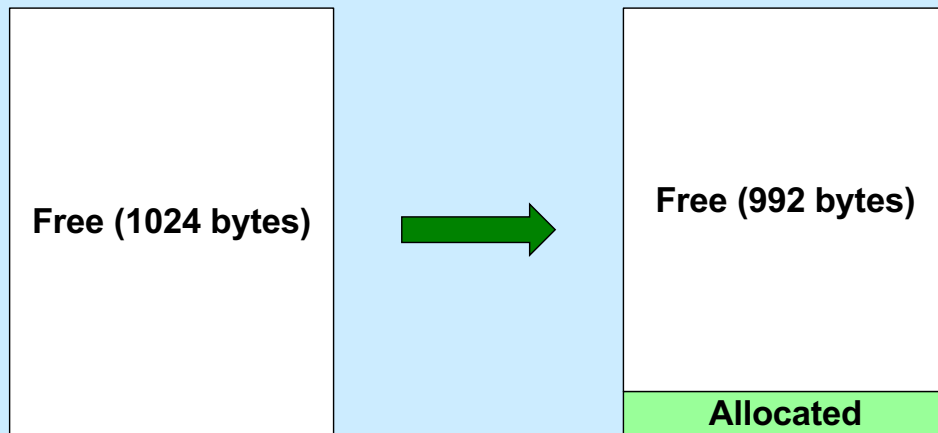


Let's assume we link together all the free blocks, as in the slide. If we'd like to allocate a block of a particular size, we need to find a free block of at least that size. What search strategy do we use to find it? An easy approach is to search, starting at the beginning of the list, until we find a block that's big enough, and use it (this is known as *first fit*). An alternative strategy, that perhaps might make better use of the available space, is to search through the entire list of free blocks and choose a block that's the smallest of those that are big enough (this is known as *best fit*).

## A Problem

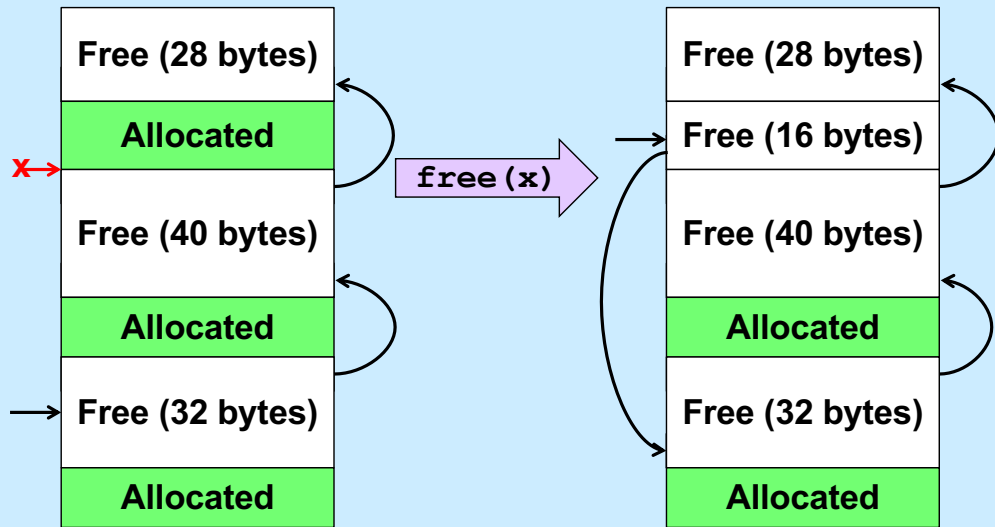
- A malloc request is for a block of 32 bytes
- The block found on the free list is 1024 bytes long
- Should malloc return a pointer to the entire 1024-byte block?

## Splitting



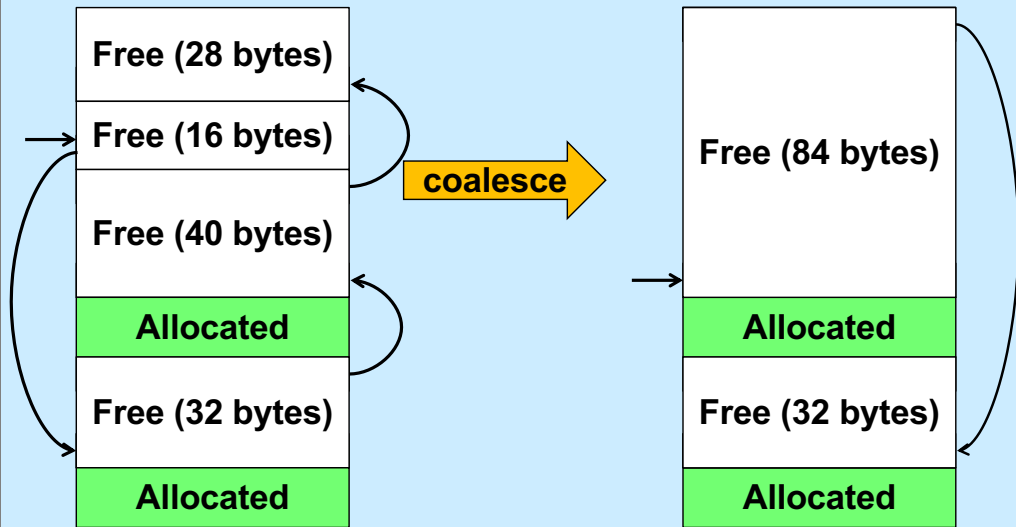
It makes no sense for malloc to return a block that's much larger than needed. Instead, it should split the block into two pieces: one piece is returned to the caller and is at least as large as was requested. The other piece is put back on the free list with an adjusted size.

## Another Problem



Here we've freed a block and end up with three free blocks in a row. The problem is that if we now attempt to allocate a block, say of size 52, we won't find a free block that's big enough, even though we clearly have enough space.

## Coalescing



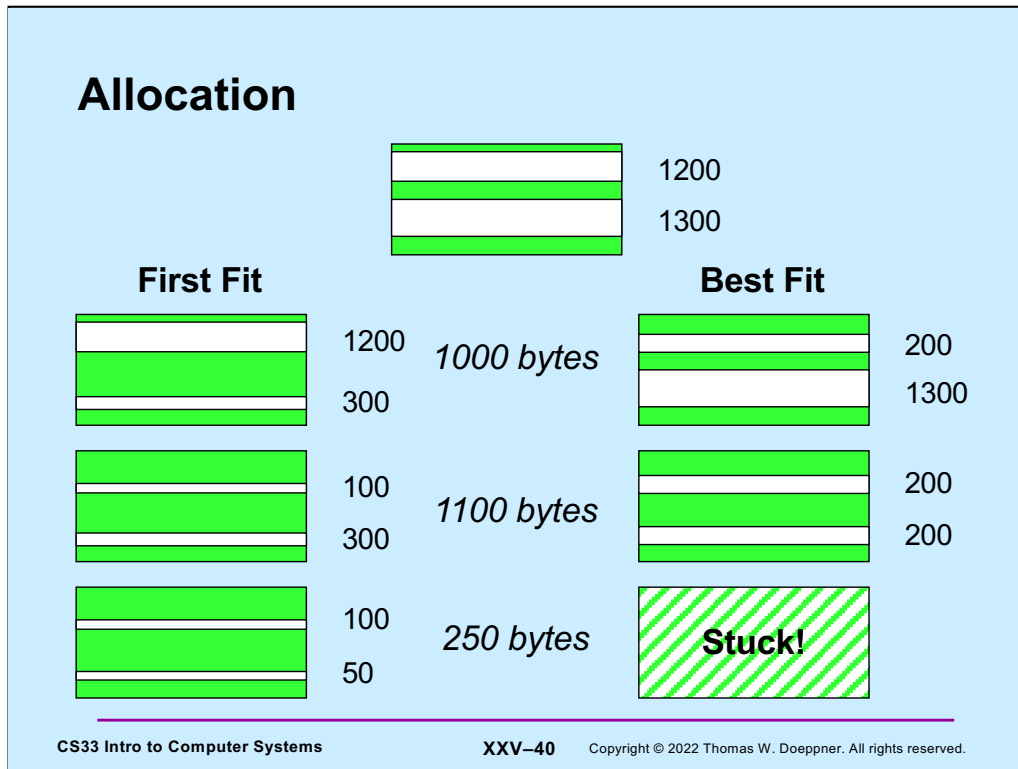
The solution is known as **coalescing**: when freeing a block, we look at adjacent blocks. if either or both adjacent blocks are free, we merge the newly freed block with its non-allocated neighbors to form a single free block whose size is the sum of the sizes of the blocks being coalesced.

## 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) first fit
- b) best fit



Consider the situation in which we have one large pool of memory from which we will allocate (and to which we will liberate) variable-sized pieces of memory. Assume that we are currently in the situation shown at the top of the picture: two unallocated areas of memory are left in the pool — one of size 1300 bytes, the other of size 1200 bytes. We wish to process a series of allocation requests, and will try out two different algorithms. The first is known as **first fit** — an allocation request is taken from the first area of memory that is large enough to satisfy the request. The second is known as **best fit** — the request is taken from the smallest area of memory that is large enough to satisfy the request. On the principle that whatever requires the most work must work the best, one might think that best fit would be the algorithm of choice.

The picture illustrates a case in which first fit behaves better than best fit. We first allocate 1000 bytes. Under the first-fit approach (shown on the left side), this allocation is taken from the topmost region of free memory, leaving behind a region of 300 bytes of still unallocated memory. With the best-fit approach (shown on the right side), this allocation is taken from the bottommost region of free memory, leaving behind a region of 200 bytes of still-unallocated memory. The next allocation is for 1100 bytes. Under first fit, we now have two regions of 300 bytes and 100 bytes. Under best fit, we have two regions of 200 bytes. Finally, there is an allocation of 250 bytes. Under first fit this leaves behind two regions of 50 bytes and 100 bytes, but the allocation cannot be handled under best fit — neither remaining region is large enough.

This example comes from the classic book, **The Art of Computer Programming, Vol. 1, Fundamental Algorithms**, by Donald Knuth.



## 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

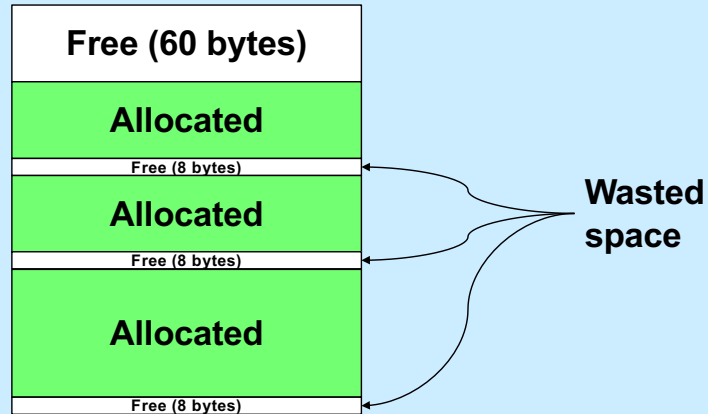
Neither first fit nor best fit is ideal. In practice, both work reasonably well in most situations. First fit has the advantage in that it doesn't always require looking at the sizes of all free blocks of memory.

# Fragmentation

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

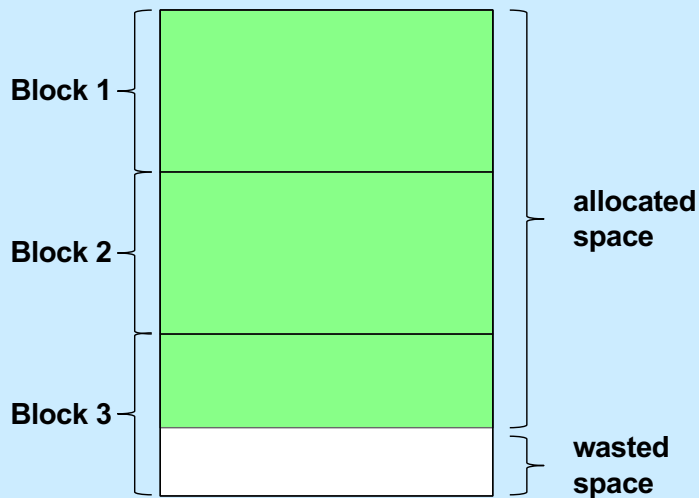
When we analyze the behavior of our storage-allocation approaches, we're concerned about **fragmentation** – how much storage is wasted.

# External Fragmentation



**External fragmentation** is when our allocation policy produces free blocks that are too small to be of use.

## Internal Fragmentation



While this isn't important for this course, internal fragmentation occurs when memory is allocated in fixed-size blocks, say 4k bytes each. If we allocate space for a data structure whose size is not a multiple of the block size, the wasted space is said to be due to **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
    - »  $O(n)$

LIFO (last in first out) insertion simply means that items are always inserted at the beginning of the free list. With ordered insertion, we keep the free list ordered by the size of the block (from smallest to largest). Note that LIFO insertion tends to put larger blocks at the beginning of the free list, which is good for first-fit allocation.

Note that for the malloc project (coming out soon), we will do first fit with LIFO insertion.

## 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) first-fit with LIFO insertion
- b) first-fit with ordered insertion
- c) best-fit

By “modest memory demands”, we mean that **malloc**, **free**, and related functions are not called frequently.