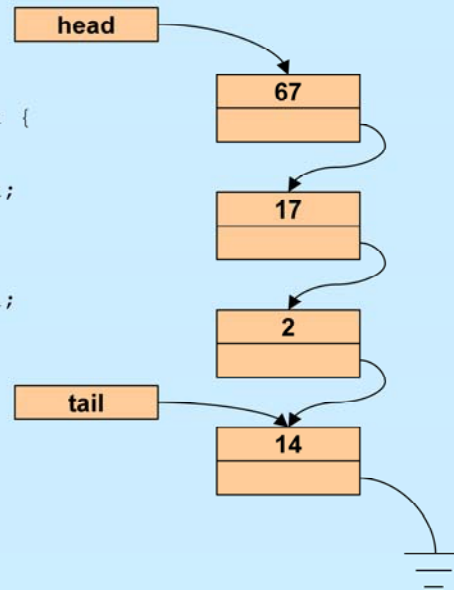


CS 33

Intro to Storage Allocation

A Queue

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



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 routine — *calloc* and *realloc*)

Enqueue

```
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 = new;
    } else {
        tail->next = new;
    }
    tail = new;
    return 1;
}
```

Dequeue

```
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;  
}
```

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. 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, possibly premature
        return buf;
    }
    if ((buf+next_read)[bytes_read-1] == '\n') {
        // end of line
        break;
    }
}
```

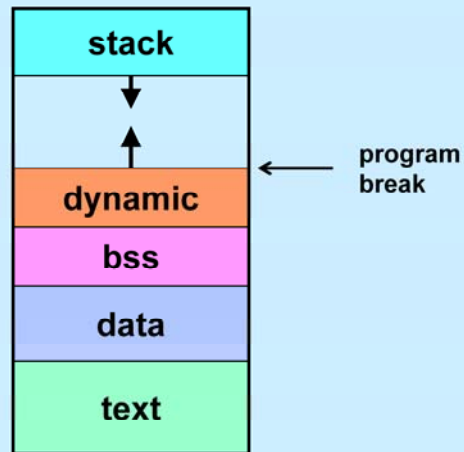

Get (contiguous) Input (3)

```
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;
}
```

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*

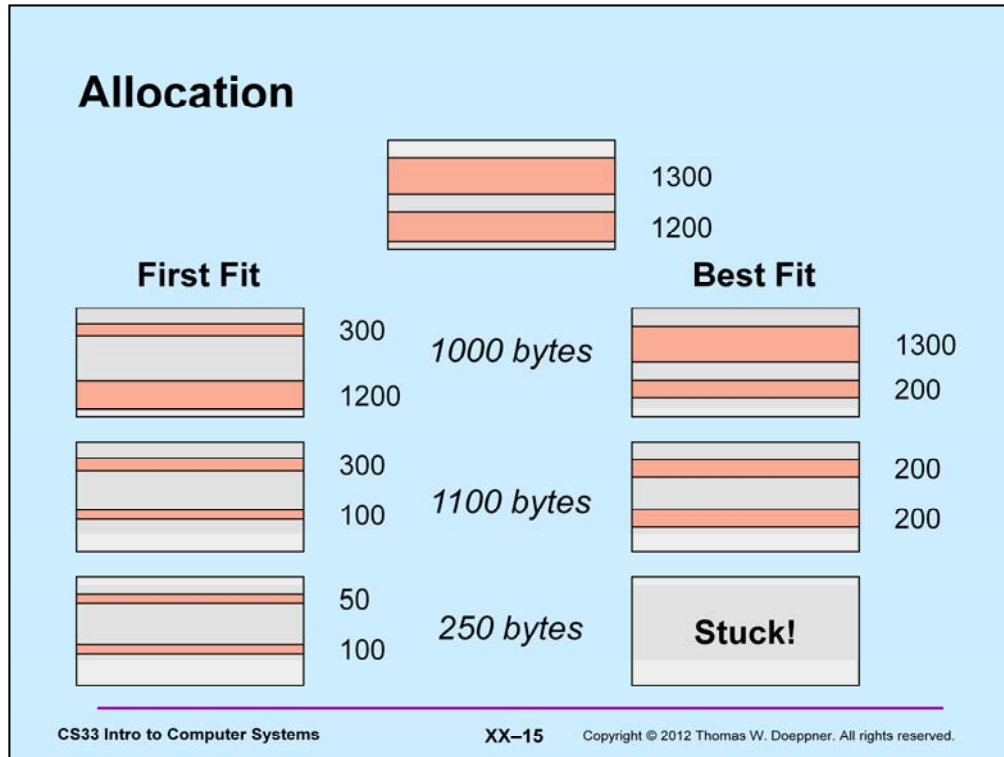
Managing Dynamic Storage

- **Strategy**
 - get a “chunk” of memory from the OS using *sbrk*
 - » 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

Dynamic Storage Allocation Overview

- **Goal:** allow dynamic creation and destruction of data structures
- **Concerns:**
 - efficient use of heap
 - efficient use of processor time
- **Example:**
 - first-fit vs. best-fit allocation

Much of the material in this and the next few slides is taken from *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, by D. Knuth.

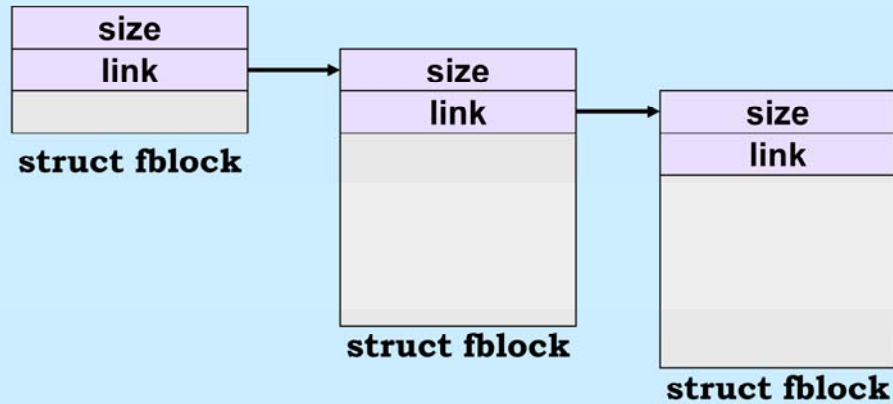


Consider the situation in which we have one large pool of memory from which we will allocate (and to which we will liberate) variable-sized pieces of memory. Assume that we are currently in the situation shown at the top of the picture: two unallocated areas of memory are left in the pool—one of size 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.

Clearly, one could come up with examples in which best fit performs better. However, simulation studies performed by Knuth have shown that, on the average, first fit works best. Intuitively, the reason for this is that best fit tends to leave behind a large number of regions of memory that are too small to be of any use.

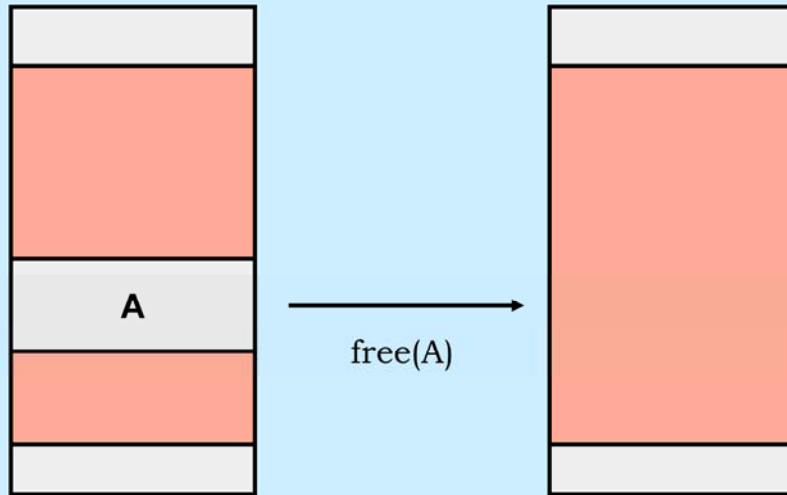
Implementing First Fit: Data Structures



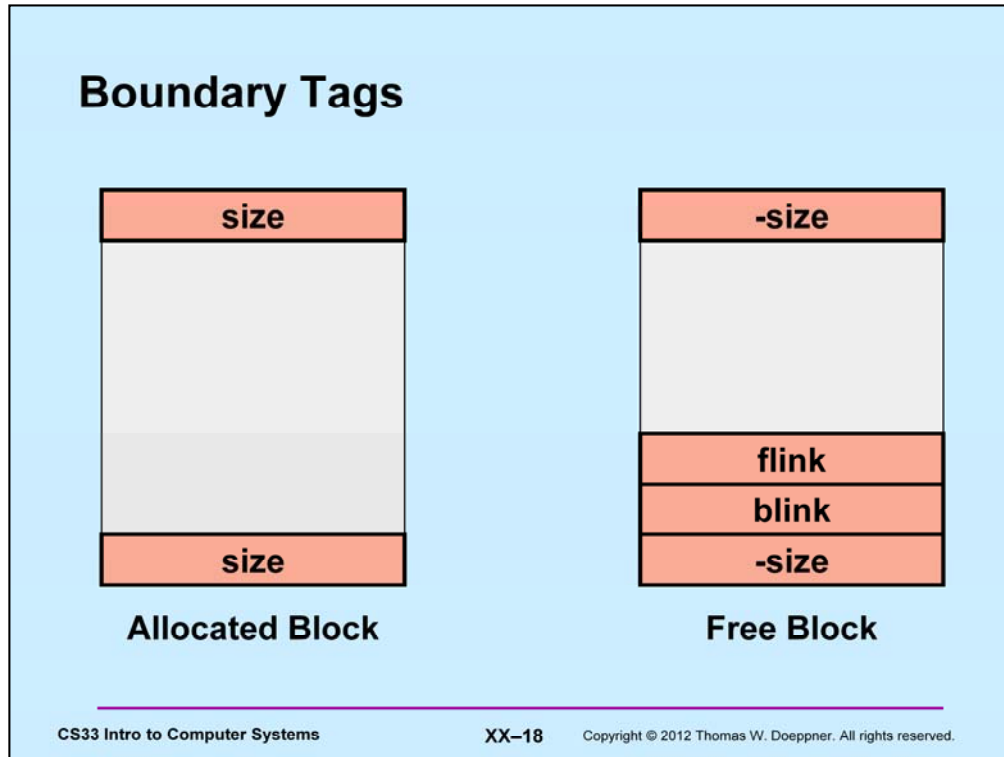
We now look at an implementation of the first-fit allocation algorithm. We need a data structure—*struct fblock*—to represent an unallocated region of memory. Since these regions are of variable size, the data structure has a *size* field. We need to link the unallocated regions together, and thus the data structure has a *link* field. Conceptually, the data structure represents the entire region of unallocated memory, but, since C has no natural ability to represent variable-sized structures, we define names for only the *size* and *link* fields.

All of the fblocks are singly linked into a free list or avail list. The header for this list is also a *struct fblock*.

Liberation of Storage

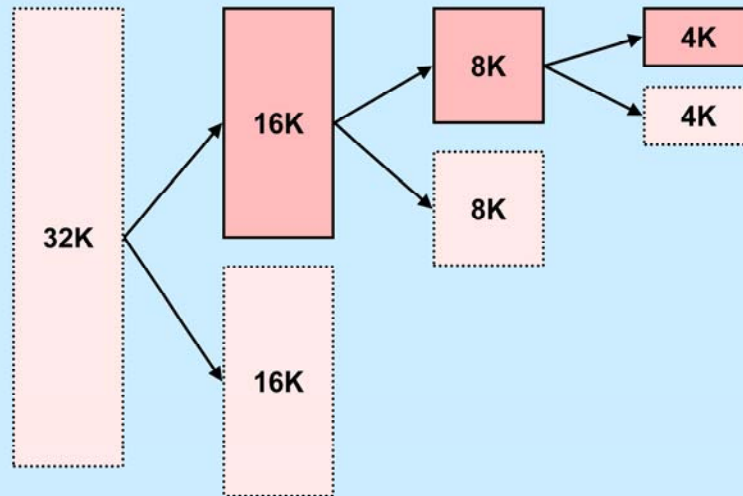


The liberation of storage is more difficult than its allocation, for the reason shown in the picture. Here the shaded regions are unallocated memory. The region of storage, A, separating the two unallocated regions is about to be liberated. The effect of doing this should be to produce one large region of unallocated storage rather than three adjacent smaller regions. Thus the liberation algorithm must be able to handle this sort of situation.



A simple method for implementing storage liberation is to use a technique known as *boundary tags*. The idea is that each region of memory, whether allocated or unallocated, has a boundary tag at each end indicating its size and whether it is allocated or not. (A positive size means allocated, a negative size means unallocated.) Thus, when we liberate a region of memory, we can quickly check the adjacent regions to determine if they too are free. Free regions are linked into a doubly linked list; thus free blocks also contain two link fields—a forward link (*flink*) and a backward link (*blink*). We call the structure representing a free block a *struct block*. (In the picture, storage addresses increase towards the top of the page, so that a pointer to a *struct block* points to the bottom of the free block.)

Buddy Lists



The buddy system is a simple dynamic storage allocation scheme that works surprisingly well. Storage is maintained in blocks whose sizes are powers of two. Requests are rounded up to the smallest such power greater than the request size. If a block of that size is free, it's taken, otherwise the smallest free block greater than the desired size is found and split in half — the two halves are called buddies. If the size of either buddy is what's needed, one of them is allocated (the other remains free). Otherwise one of the buddies is split in half. This splitting continues until the appropriate size is reached.

Liberating a block now is easy: if the block's buddy is free, you join the block being liberated with its buddy, forming a larger free block. If this block's buddy is free, you join the two of them, and so forth until the largest free block possible is formed.

One bit in each block, say in its first or last byte, is used as a tag to indicate whether the block is free. Determining the address of a block's buddy is simple. If the block is of size 2^k , then the rightmost $k-1$ bits of its address are zeros. The next bit (to the left) is zero in one buddy and one in the other; so you simply complement that bit to get the buddy's address.