15-213

"The course that gives CMU its Zip!"

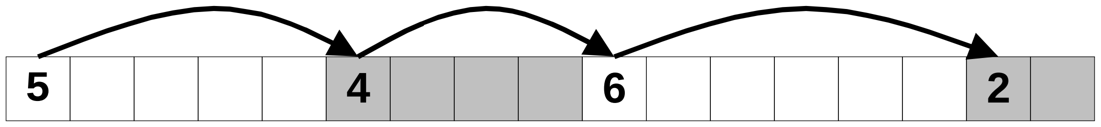
Dynamic Memory Allocation II
Nov 7, 2000

Topics

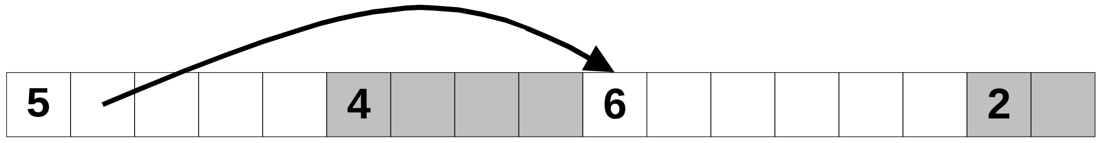
- doubly-linked free lists
- segregated free lists
- garbage collection
- memory-related perils and pitfalls

Keeping track of free blocks

- ***Method 1***: implicit list using lengths -- links all blocks

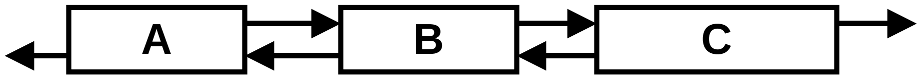


- ***Method 2***: explicit list among the free blocks using pointers within the free blocks



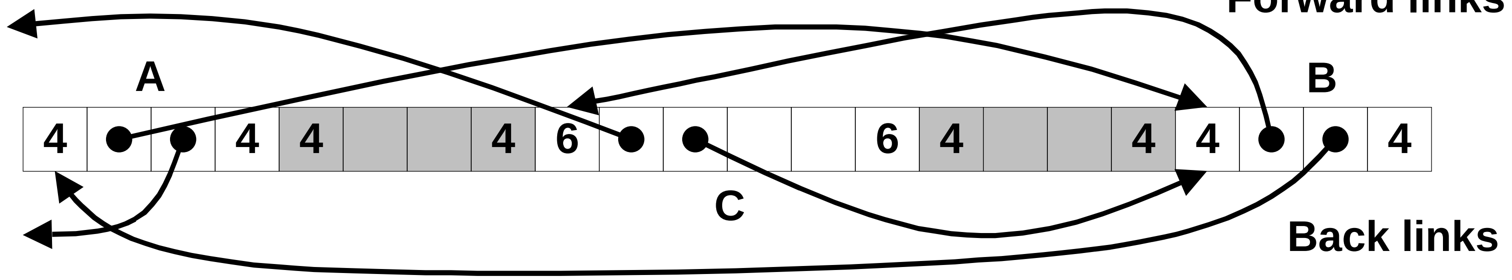
- ***Method 3***: segregated free lists
  - Different free lists for different size classes
- ***Method 4***: blocks sorted by size (not discussed)
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Explicit free lists



Use data space for link pointers

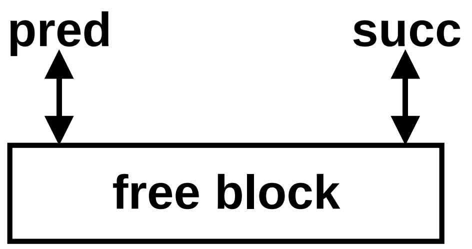
- Typically doubly linked
- Still need boundary tags for coalescing



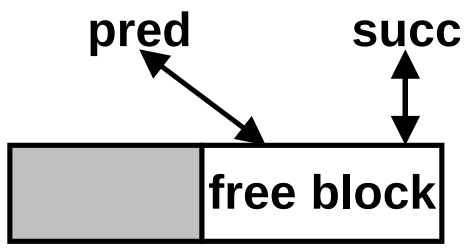
- It is important to realize that links are not necessarily in the same order as the blocks

Allocating from explicit free lists

Before:



After:
(with splitting)

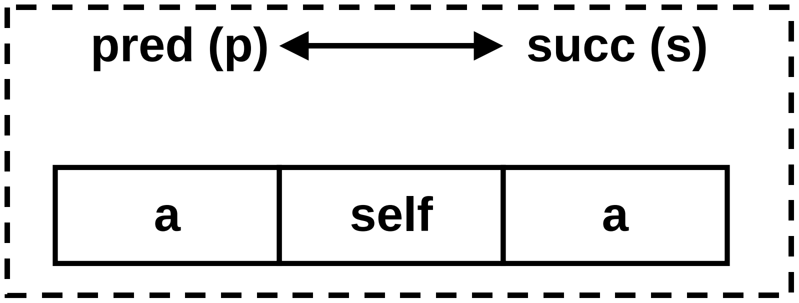


Freeing with explicit free lists

***Insertion policy*: Where to put the newly freed block in the free list**

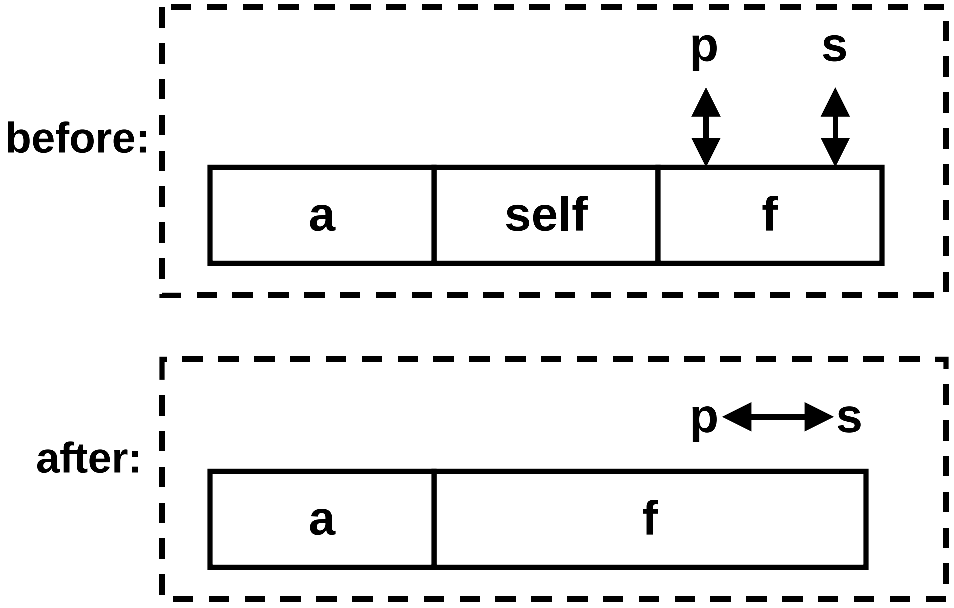
- **LIFO (last-in-first-out) policy**
  - insert freed block at the beginning of the free list
  - pro: simple and constant time
  - con: studies suggest fragmentation is worse than address ordered.
- **Address-ordered policy**
  - insert freed blocks so that free list blocks are always in address order
    - » i.e. addr(pred) < addr(curr) < addr(succ)
  - con: requires search
  - pro: studies suggest fragmentation is better than LIFO

Freeing with a LIFO policy



Case 1: a-a-a

- **insert self at beginning of free list**



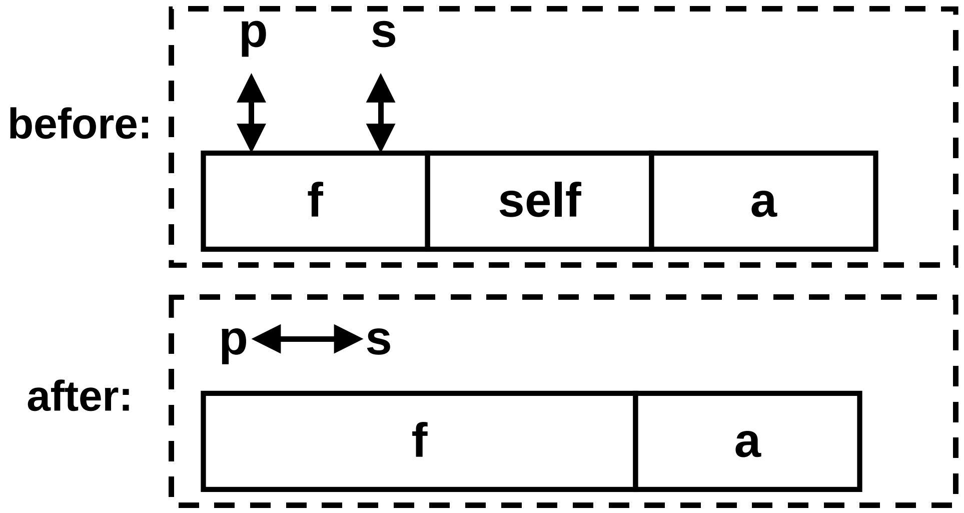
Case 2: a-a-f

- **splice out next, coalesce self and next, and add to beginning of free list**

Freeing with a LIFO policy (cont)

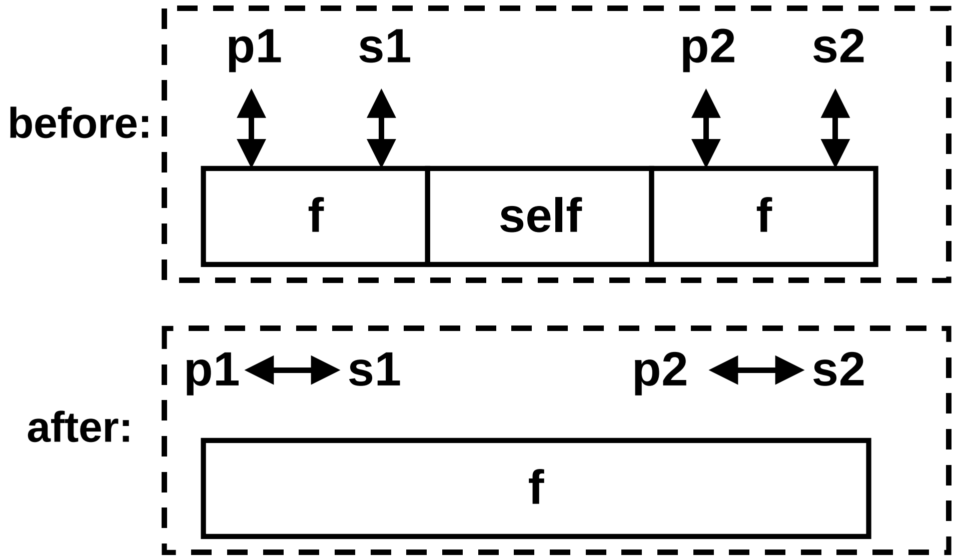
Case 3: f-a-a

- splice out prev, coalesce with self, and add to beginning of free list



Case 4: f-a-f

- splice out prev and next, coalesce with self, and add to beginning of list



Explicit list summary

Comparison to implicit list:

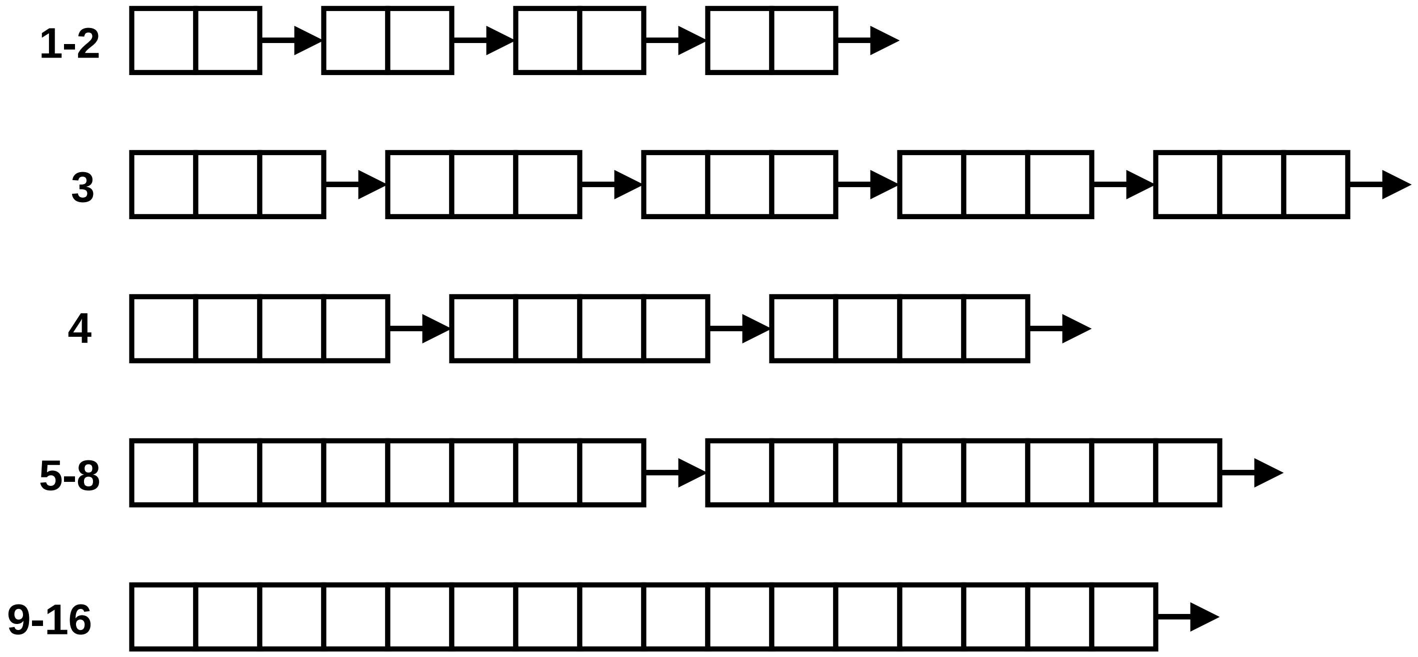
- **Allocate is linear time in number of free blocks instead of total blocks -- much faster allocates when most of the memory is full**
- **Slightly more complicated allocate and free since needs to splice blocks in and out of the list**
- **Some extra space for the links (2 extra words needed for each block)**

Main use of linked lists is in conjunction with segregated free lists

- **Keep multiple linked lists of different size classes, or possibly for different types of objects**

Segregated Storage

Each size “class” has its own collection of blocks



- Often have separate collection for every small size (2,3,4,…)
- For larger sizes typically have a collection for each power of 2

Simple segregated storage

Separate heap and free list for each size class

No splitting

To allocate a block of size n:

- **if free list for size n is not empty,**
  - allocate first block on list (note, list can be implicit or explicit)
- **if free list is empty,**
  - get a new page
  - create new free list from all blocks in page
  - allocate first block on list
- **constant time**

To free a block:

- **Add to free list**
- **If page is empty, return the page for use by another size (optional)**

Tradeoffs:

- **fast, but can fragment badly**

Segregated fits

Array of free lists, each one for some size class

To allocate a block of size n:

- **search appropriate free list for block of size m > n**
- **if an appropriate block is found:**
  - split block and place fragment on appropriate list (optional)
- **if no block is found, try next larger class**
- **repeat until block is found**

To free a block:

- **coalesce and place on appropriate list (optional)**

Tradeoffs

- **faster search than sequential fits (i.e., log time for power of two size classes)**
- **controls fragmentation of simple segregated storage**
- **coalescing can increase search times**
  - deferred coalescing can help

For more information of dynamic storage allocators

D. Knuth, “The Art of Computer Programming, Second Edition”, Addison Wesley, 1973

- the classic reference on dynamic storage allocation

Wilson et al, “Dynamic Storage Allocation: A Survey and Critical Review”, Proc. 1995 Int’l Workshop on Memory Management, Kinross, Scotland, Sept, 1995.

- comprehensive survey
- available from the course web page (see Documents page)

Implicit Memory Management Garbage collector

Garbage collection: automatic reclamation of heap-allocated storage -- application never has to free

```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

Common in functional languages, scripting languages, and modern object oriented languages:

- Lisp, ML, Java, Perl, Mathematica,

Variants (conservative garbage collectors) exist for C and C++

- Cannot collect all garbage

Garbage Collection

How does the memory manager know when memory can be freed?

- **In general we cannot know what is going to be used in the future since it depends on conditionals**
- **But we can tell that certain blocks cannot be used if there are no pointers to them**

Need to make certain assumptions about pointers

- **Memory manager can distinguish pointers from non-pointers**
- **All pointers point to the start of a block**
- **Cannot hide pointers (e.g. by coercing them to an int, and then back again)**

Classical GC algorithms

Mark and sweep collection (McCarthy, 1960)

- **Does not move blocks (unless you also "compact")**

Reference counting (Collins, 1960)

- **Does not move blocks (not discussed)**

Copying collection (Minsky, 1963)

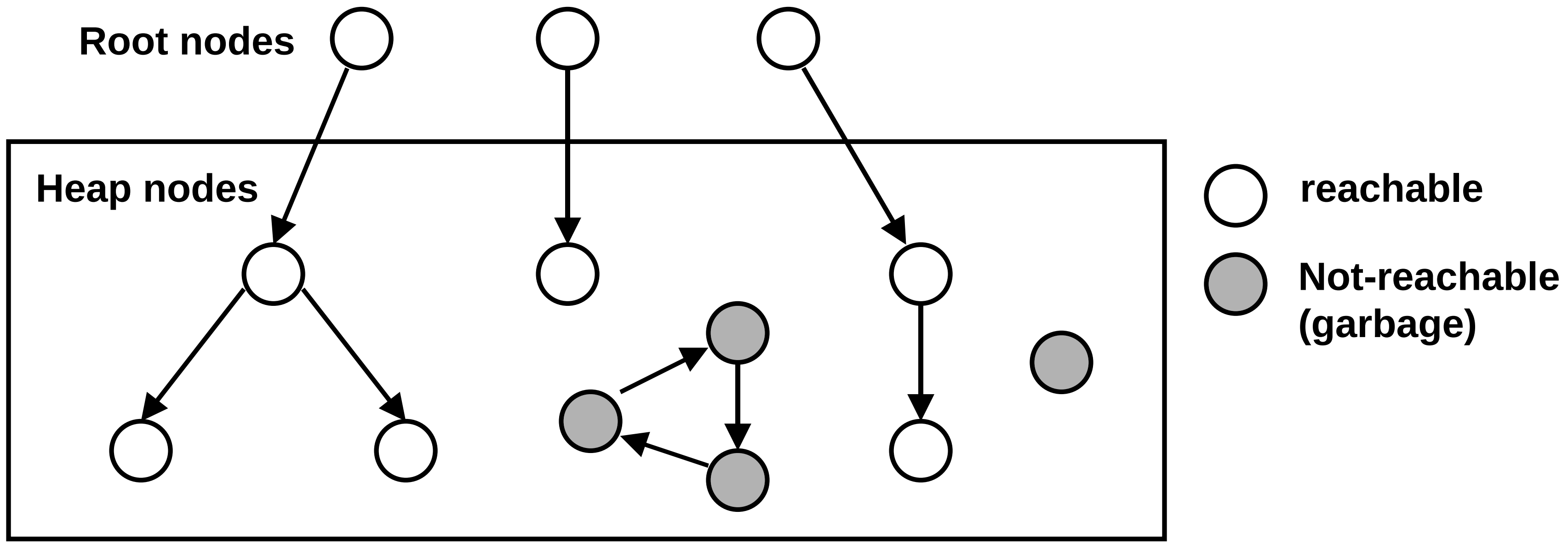
- **Moves blocks (not discussed)**

For more information see *Jones and Lin, "Garbage Collection: Algorithms for Automatic Dynamic Memory", John Wiley & Sons, 1996.*

Memory as a graph

We view memory as a directed graph

- **Each block is a node in the graph**
- **Each pointer is an edge in the graph**
- **Locations not in the heap that contain pointers into the heap are called *root* nodes (e.g. registers, locations on the stack, global variables)**



A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (never needed by the application)

Assumptions for this lecture

Application

- `new(n)`: returns pointer to new block with all locations <u>cleared</u>
- `read(b,i)`: read location `i` of block `b into register`
- `write(b,i,v)`: write v into location `i` of block `b`

Each block will have a header word

- addressed as `b[-1]`, for a block `b`
- Used for different purposes in different collectors

Instructions used by the Garbage Collector

- `is_ptr(p)`: determines whether `p` is a pointer
- `length(b)`: returns the length of block `b`, not including the header
- `get_roots()`: returns all the roots

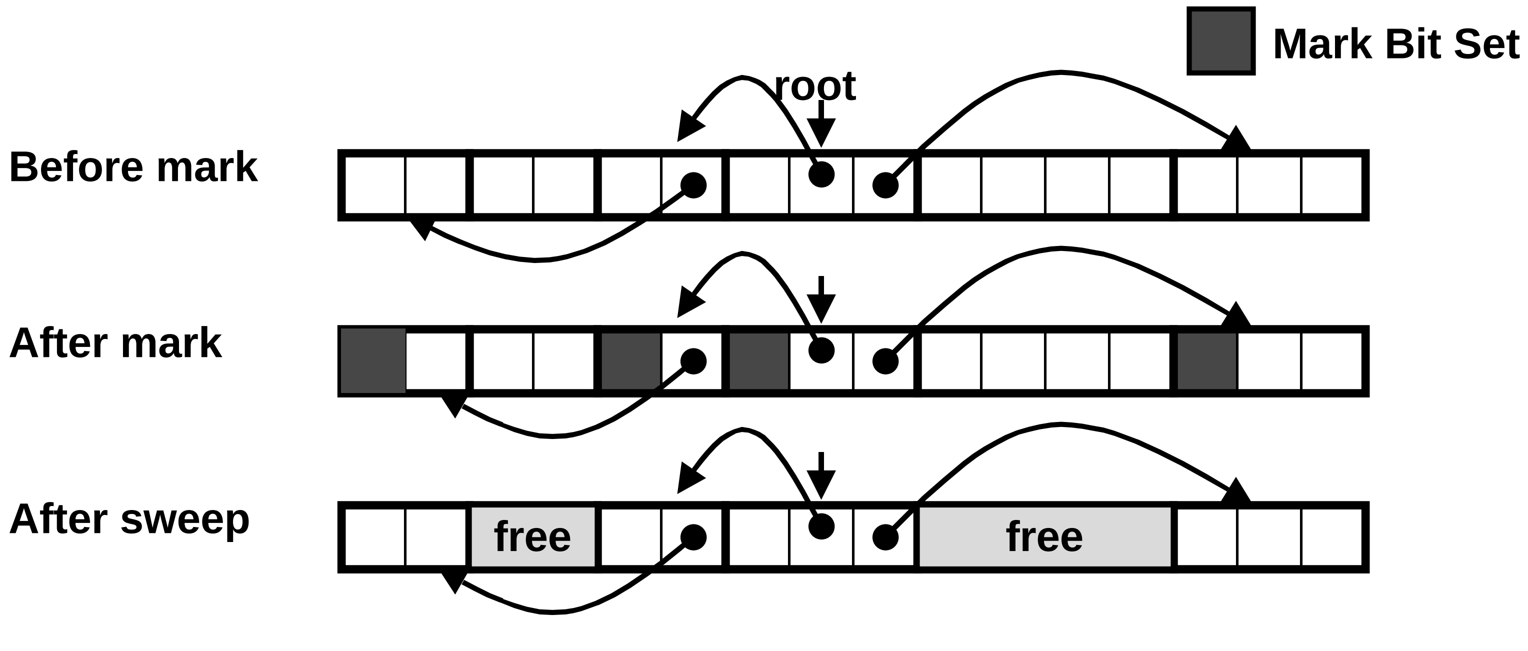
Mark and sweep collecting

Can build on top of malloc/free package

- Allocate using **malloc** until you “run out of space”

When out of space:

- Use extra **“mark bit”** in the head of each block
- **Mark:** Start at roots and set **mark bit** on all reachable memory
- **Sweep:** Scan all blocks and **free** blocks that are **not marked**



Mark and sweep (cont.)

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return        // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)  // mark all children
     mark(p[i]);
   return;
}
```

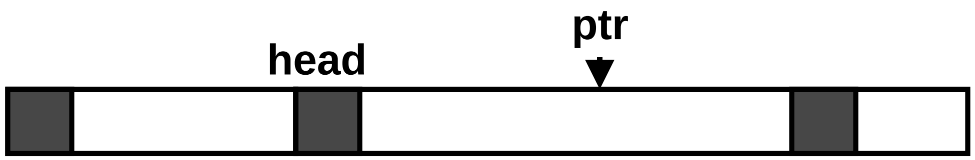
Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```

Mark and sweep in C

A C Conservative Collector

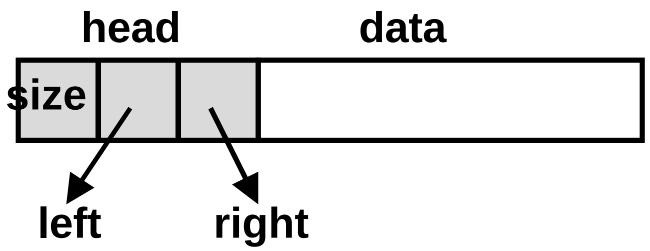
- **`Is_ptr()` determines if a word is a pointer by checking if it points to an allocated block of memory.**
- **But, in C pointers can point to the middle of a block.**



So how do we find the beginning of the block

Can use balanced tree to keep track of all allocated blocks where the key is the location

Balanced tree pointers can be stored in head (use two additional words)



Memory-related bugs

Dereferencing bad pointers

Reading uninitialized memory

Overwriting memory

Referencing nonexistent variables

Freeing blocks multiple times

Referencing freed blocks

Failing to free blocks

Dereferencing bad pointers

The classic scanf bug

```
scanf(“%d”, val);
```

Reading uninitialized memory

Assuming that heap data is initialized to zero

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = 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

```
int **p;

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

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

Overwriting memory

Off-by-one

```
int **p;

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

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

Overwriting memory

Not checking the max string size

```
char s[8];
int i;

gets(s);  /* reads “123456789” from stdin */
```

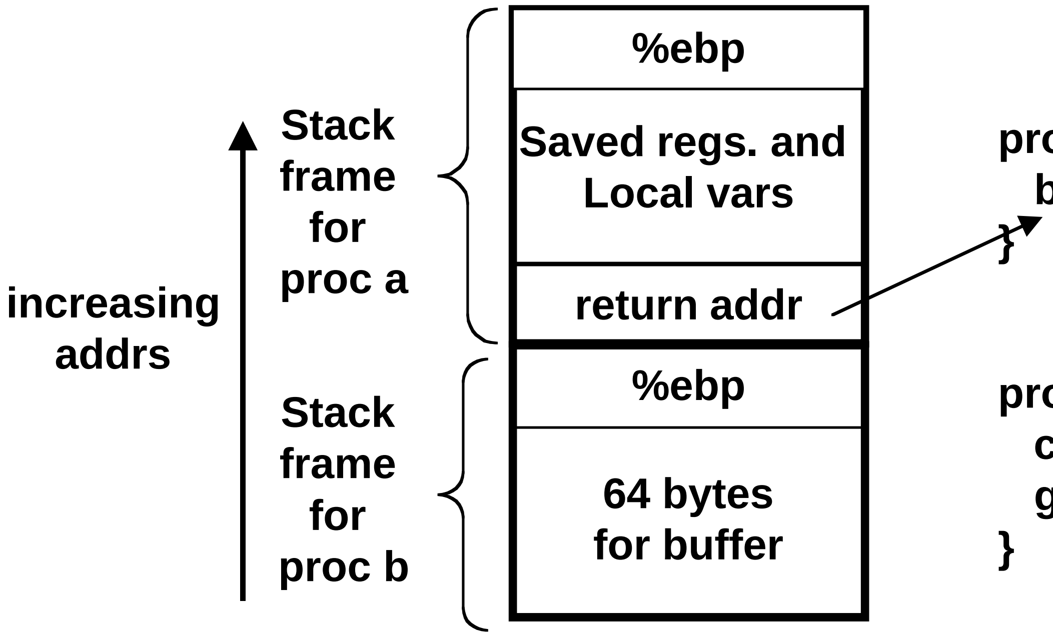
Basis for classic ***buffer overflow attacks***

- 1988 Internet worm
- modern attacks on Web servers
- AOL/Microsoft IM war

Buffer overflow attacks

Description of hole:

- **Servers that use C library routines such as gets() that don't check input sizes when they write into buffers on the stack.**
- **The following description is based on the IA32 stack conventions. The details will depend on how the stack is organized, which varies between compilers and machines**



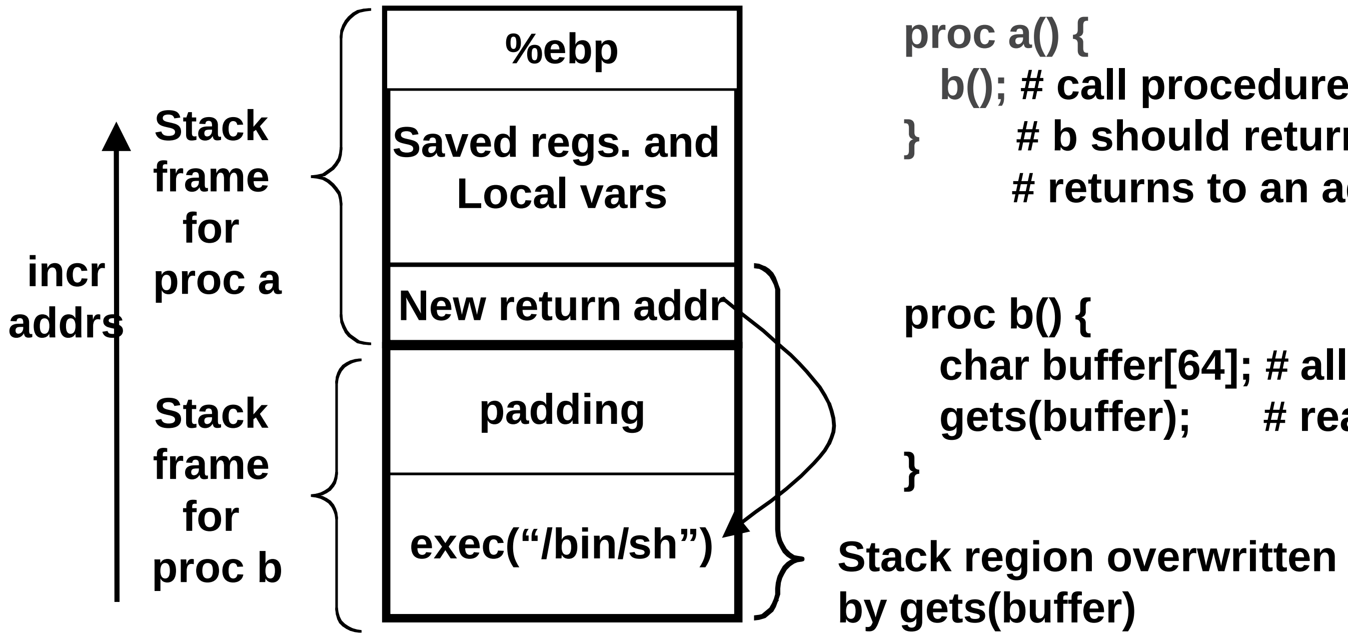
```
proc a() {
   b(); # call procedure b
}

proc b() {
   char buffer[64]; # alloc 64 bytes on stack
   gets(buffer);      # read STDIN line into buf
}
```

Buffer overflow attacks

Vulnerability stems from possibility of the gets() routine overwriting the return address for b.

- **overwrite stack frame with**
  - machine code instruction(s) that execs a shell
  - a bogus return address to the instruction



```
proc a() {
   b(); # call procedure b
}        # b should return here, instead it
         # returns to an address inside of buffer

proc b() {
   char buffer[64]; # alloc 64 bytes on stack
   gets(buffer);      # read STDIN line to buffer
}
```

Overwriting memory

Referencing a pointer instead of the object it points to

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

Overwriting memory

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

Nasty!

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

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

Referencing freed blocks

Evil!

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

Failing to free blocks (memory leaks)

slow, long-term killer!

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   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;
   <create and manipulate the rest of the list>
   ...
   free(head);
   return;
}
```

Dealing with memory bugs

Conventional debugger (gdb)

- **good for finding bad pointer dereferences**
- **hard to detect the other memory bugs**

Debugging malloc (CSRI UToronto malloc)

- **wrapper around conventional malloc**
- **detects memory bugs at malloc and free boundaries**
  - memory overwrites that corrupt heap structures
  - some instances of freeing blocks multiple times
  - memory leaks
- **Cannot detect all memory bugs**
  - overwrites into the middle of allocated blocks
  - freeing block twice that has been reallocated in the interim
  - referencing freed blocks

Dealing with memory bugs (cont.)

Binary translator (Atom, Purify)

- **powerful debugging and analysis technique**
- **rewrites text section of executable object file**
- **can detect all errors as debugging malloc**
- **can also check each individual reference at runtime**
  - bad pointers
  - overwriting
  - referencing outside of allocated block

Garbage collection (Boehm-Weiser Conservative GC)

- **let the system free blocks instead of the programmer.**