The Hardware/Software Interface

CSE351 Winter 2013

Memory Allocation III

Implicit Memory Allocation: Garbage Collection

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

```
void foo() {
   int *p = (int *)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++**
  - However, cannot necessarily collect all garbage

Garbage Collection

- **How does the memory allocator 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 <u>no pointers to them</u>

- **So the memory allocator needs to know what is a pointer and what is not – how can it do this?**

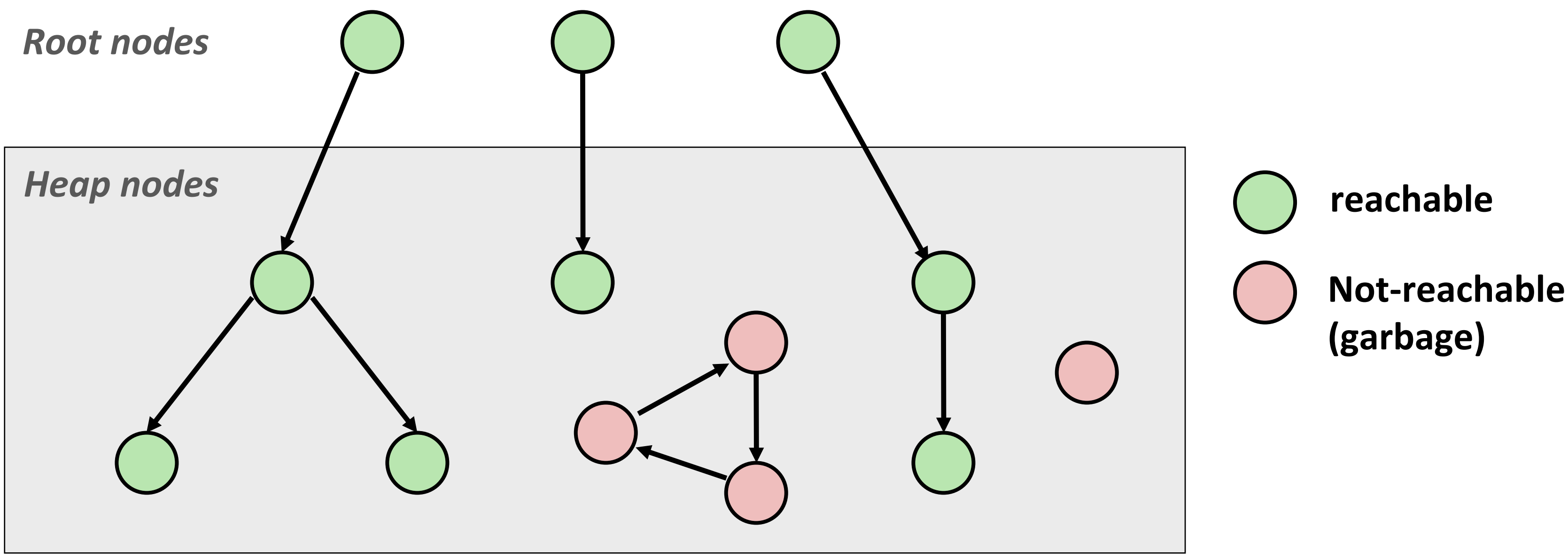
- **We'll make some assumptions about pointers:**
  - Memory allocator can distinguish pointers from non-pointers
  - All pointers point to the start of a block in the heap
  - Application cannot hide pointers
    (e.g., by coercing them to an `int`, and then back again)

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)
- **Generational Collectors (Lieberman and Hewitt, 1983)**
  - Collection based on lifetimes
    - Most allocations become garbage very soon
    - So focus reclamation work on zones of memory recently allocated
- **For more information:**
  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 allocated heap 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)

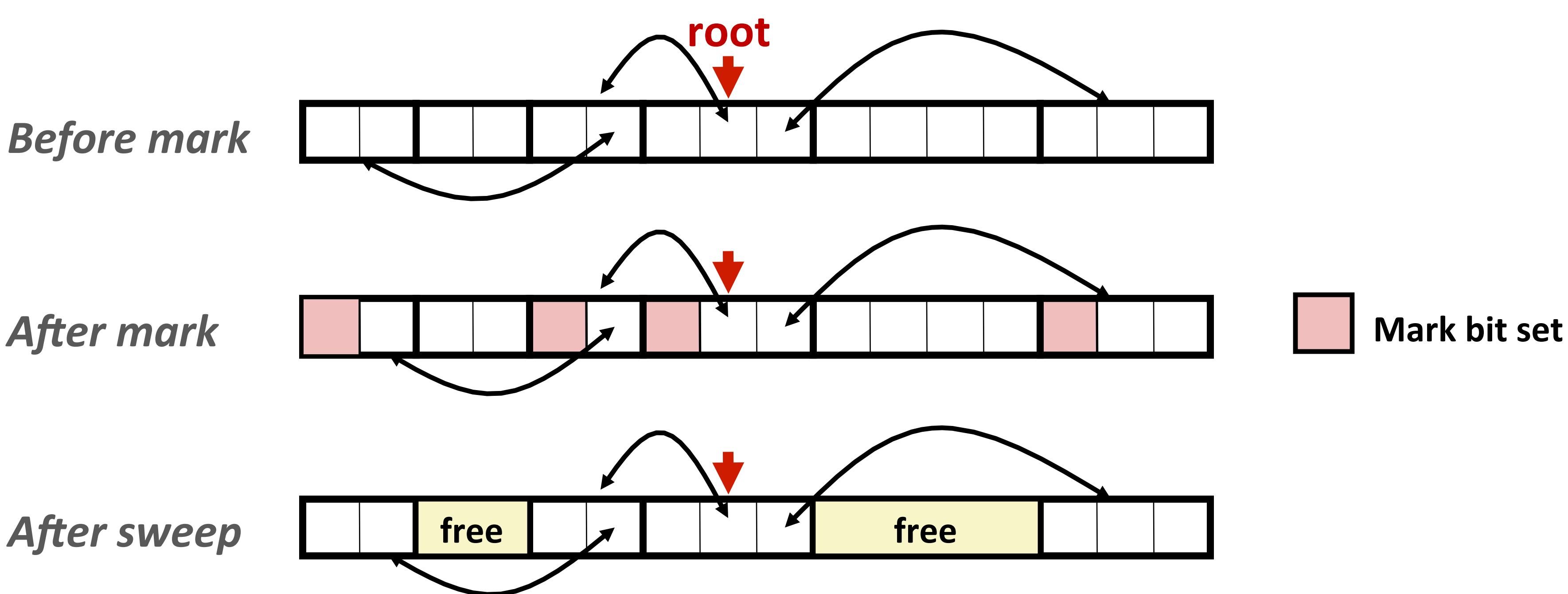


A node (block) is *reachable* if there is a path from any root to that node

Non-reachable nodes are *garbage* (cannot be needed by the application)

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 each reachable block
  - ***Sweep:*** Scan all blocks and free blocks that are not marked



Assumptions For a Simple Implementation

- **Application can use functions such as:**
  - **`new(n)`** : returns pointer to new block with all locations cleared
  - **`read(b,i)`**: read location **`i`** of block **`b`** into register
    - **`b[i]`**
  - **`write(b,i,v)`**: write **`v`** into location **`i`** of block **`b`**
    - **`b[i] = v`**
- **Each block will have a header word**
    - **`b[-1]`**

- **Functions used by the garbage collector:**
  - **`is_ptr(p)`**: determines whether **`p`** is a pointer to a block
  - **`length(p)`**: returns length of block pointed to by **`p`**, not including header
  - **`get_roots()`**: returns all the roots

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

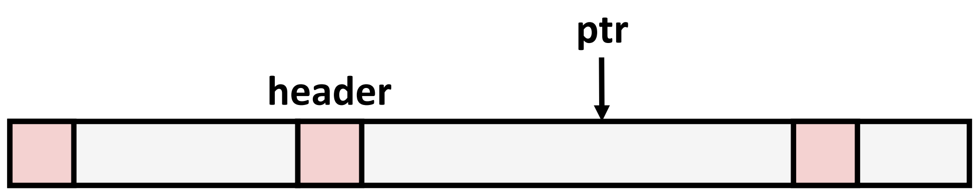
```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // do nothing if not pointer
   if (markBitSet(p)) return;       // check if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // recursively call mark on
     mark(p[i]);              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

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                // while not at end of heap
      if markBitSet(p)              // check if block is marked
         clearMarkBit();            // if so, reset mark bit
      else if (allocateBitSet(p))   // if not marked, but allocated
         free(p);                   // free the block
      p += length(p);               // adjust pointer to next block
}
```

Conservative Mark & Sweep in C

- **Would mark & sweep work in C?**
  - **`is_ptr()`** (previous slide) determines if a word is a pointer by checking if it points to an allocated block of memory
  - But in C, pointers can point into the *middle* of allocated blocks (not so in Java)
    - Makes it tricky to find all allocated blocks in mark phase



  - There are ways to solve/avoid this problem in C, but the resulting garbage collector is *conservative:*
    - Every reachable node correctly identified as reachable, but some unreachable nodes might be incorrectly marked as reachable

Memory-Related Perils and Pitfalls

- 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**

```
int val;

...

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

- **Will cause `scanf` to interpret contents of `val` as an address!**
  - Best case: program terminates immediately due to segmentation fault
  - Worst case: contents of **`val`** correspond to some valid read/write area of virtual memory, causing **`scanf`** to overwrite that memory, with disastrous and baffling consequences much later in program execution

Reading Uninitialized Memory

- Assuming that heap data is initialized to zero

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

```
int **p;

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

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

Overwriting Memory

- **Off-by-one error**

```
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**
  - Your lab assignment #3

Overwriting Memory

- **Misunderstanding pointer arithmetic**

```
int *search(int *p, int val) {

   while (p && *p != val)
      p += sizeof(int);

   return p;
}
```

Overwriting Memory

- Referencing a pointer instead of the object it points to

```
int *getPacket(int **packets, int *size) {
   int *packet;
   packet = packets[0];
   packets[0] = packets[*size - 1];
   *size--;   // what is happening here?
   reorderPackets(packets, *size);
   return(packet);
}
```

- ‘--’ and ‘*’ operators have same precedence and associate from right-to-left, so -- happens first!

Referencing Nonexistent Variables

- **Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```

Freeing Blocks Multiple Times

- Nasty!

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

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

- What does the free list look like?

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

Referencing Freed Blocks

- **Evil!**

```
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)

- **Slow, silent, long-term killer!**

```
foo() {
   int *x = (int *)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 =
      (struct list *)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` (UToronto CSRI `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.)

- **Some malloc implementations contain checking code**
  - Linux glibc malloc: `setenv MALLOC_CHECK_ 2`
  - FreeBSD: `setenv MALLOC_OPTIONS AJR`
- **Binary translator: valgrind (Linux), 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