Announcements

- **Program #5 is on the web**
  - correction to the .o files made on Monday
  - can copy just that
- **Regrade requests on Exam #2**
  - Due by Thursday
  - Must be entered into grades.cs.umd.edu
  - Must submit paper

- **Reading**
  - Bryant & O’Hallaron 10.10 (Tuesday)

---

Garbage Collection

- **What is Garbage?**
  - memory the program can't get to anymore
  - Example:
    ```
    ptr = malloc(40);  ptr = malloc(40);
    ptr = NULL;        ptr2 = ptr;
    ptr = NULL;
    ```

- **How to detect Garbage?**
  - Need to know what is a pointer
    - know what everything is (Java)
    - guess: anything that looks like a pointer is one (C)
      - any 32-bit quantity in globals, on stack, or in heap is assumed to be a pointer.
      - anything in a memory region in a valid range
      - anything that could be pointed to – is non-garbage
Approach 1: Reference Counting

- Old technique (1960)
- Each object has count of number of pointers to it from other objects and from the stack
  - When count reaches 0, object can be deallocated
- Counts tracked by either compiler or manually
- To find pointers, need to know layout of objects
  - In particular, need to distinguish pointers from ints
- Method works mostly for reclaiming memory; doesn’t handle fragmentation problem

Reference Counting Example

```
stack 1
```

```
     1

     1

     1
```
Reference Counting Example

```
stack
```

Reference Counting Example

```
stack
```
Reference Counting Example

stack

Reference Counting Example

stack
Approach 2: Mark & Sweep

- **Figure out what is reachable**
  - Memory that is reachable is not garbage
- **Compute reachability graph**
  - Follow all pointers
  - Figure out what memory can be reached
  - Anything that is not reachable is garbage
    - Collect it

Reachability Graph

- Not-reachable (garbage)
- Reachable
- Variables (Inside the program)
**Mark and Sweep Garbage Collection**

- **Mark Function**
  - each allocated block has a "marked" field
  - isPtr returns header for ptr (if in heap) or null

```c
void mark(ptr p) {
    header * b;
    b = isPtr(p);
    if (!b) return;
    if (b->marked) return;
    b->marked = 1;
    for (i=0; i < b->size; i++) {
        mark(b + i + 1); /* assumes heap header is 32 bits */
    }
}
```
Mark & Sweep

- **Sweep**
  
  ```c
  void sweep(header *curr) {
    while (curr->size) {
      if (curr->marked) {
        curr->marked = 0;
      } else if (curr->allocated) {
        free(curr+1);
      }
      curr = curr->next;
    }
  }
  ```

- **Calling Mark and Sweep**
  - Call when the allocate function is about to call sbrk

What ptrs to pass to Mark?

- **Need to figure out non-heap pointers**
  - Global memory
  - Local memory (stack)

```c
for (i in the set of global and local variables) {
    mark(i);
}
```
Mark & Sweep Issues

- **Writing isPtr**
  - need to be able to find a block header given a ptr into it
  - Options:
    - Balanced tree of memory heap regions
      - search tree to find which region point is in
    - Use segregated heaps
      - power of 2 sized
      - keep a table of which heaps are where
      - use modular arithmetic to find header

Issues in Garbage Collection

- **In language like C**
  - Something may look like a pointer, but not be one
    - Conservative Garbage Collector
      - Never frees memory that is in use
      - Sometimes leaves some memory un-collected
Tradeoffs with Mark and Sweep

- **Pros:**
  - No problem with cycles
  - Memory writes have no cost

- **Cons:**
  - Fragmentation
    - Available space broken up into many small pieces
      - Thus many mark-and-sweep systems may also have a *compaction* phase (like defragmenting your disk)
  - Cost proportional to heap size
    - Sweep phase needs to traverse whole heap — it touches dead memory to put it back on to the free list
  - Not appropriate for real-time applications
    - You wouldn’t like your auto’s braking system to stop working for a GC while you are trying to stop at a busy intersection

Approach 3: Stop and Copy GC

- **Like mark and sweep, but only touches live objects**
  - Divide heap into two equal parts (semispaces)
  - Only one semispace active at a time
  - At GC time, flip semispaces
    - Trace the live data starting from the stack
    - Copy live data into other semispace
    - Declare everything in current semispace dead; switch to other semispace
Stop and Copy Example

Stack

Stop and Copy Example

Stack
Stop and Copy Example

1. Stack
2. Copy
3. Stop
Stop and Copy Example

Stop and Copy Tradeoffs

- **Pros:**
  - Only touches live data
  - No fragmentation; automatically compacts
    - Will probably increase locality
- **Cons:**
  - Requires twice the memory space
  - Like mark and sweep, need to “stop the world”
    - Program must stop running to let garbage collector move around data in the heap
Java

- Bimodal distribution of duration of existence
- Java makes use of this hierarchy
  - things created for short term are put in one place
  - after it survives one round, it is moved to the more permanent space
  - after it reaches the highest level, it stays around

- Different Techniques in Different Spaces
  - can run in the background
  - implementations that do need to complete – must restart if another higher priority thread is activated
  - not all types need to complete before other processes

Good or Bad
If Bad – why?

```c
int *f1(int a){
    int x = a* 2;
    return &x;
}
```

```c
int *f1(int a){
    int x = a* 2;
    int *p = &x;
    return p;
}
```

```c
int *f1(int a){
    int x = a* 2;
    a = x - 99;
    return &a;
}
```
More Good or Bad & Bad – why?

```c
int **makearray1(int n, int m){
    int j;
    int **A = (int **)malloc(n*sizeof(int));
    for (j = 0; j < n; j++){
        A[j] = (int *) malloc (m*sizeof(int));
        return A;
    }
}
```

```c
int **makearray1(int n, int m){
    int j;
    int **A = (int **)malloc(n*sizeof(int));
    for (j = 0; j <= n; j++){
        A[j] = (int *) malloc (m*sizeof(int));
    }
    return A;
}
```

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