Dynamic Memory Allocation: Advanced Concepts

CSC 235 - Computer Organization

References

- Slides adapted from CMU

Review: Dynamic Memory Allocation

- Programmers use dynamic memory allocators (such as `malloc`) to acquire virtual memory (VM) at run time.
  - For data structures where the size is only known at runtime
- Dynamic memory allocators manage an area of process VM known as the heap.

Review: Keeping Track of Free Blocks

- Method 1: Implicit list using length; links all blocks
  - Need to tag each block as allocated/free
- Method 2: Explicit list among the free blocks using pointers
  - Need space for pointers
- Method 3: Segregated free list
  - Different free lists for different size classes
- Method 4: Blocks sorted by size
  - Can use a balanced tree with pointers within each free block, and the length used as a key

Review: Implicit Lists Summary

- Implementation: very simple
- Allocate cost: linear time worst case
- Free cost: constant time worst case (even with coalescing)
- Memory overhead: depends on placement policy
- Not used in practice for `malloc`/`free` because of linear time allocation
- The concepts of splitting and boundary tag coalescing are general to all allocators

Explicit Free Lists

- Maintain list(s) of *free* blocks, not *all* blocks
  - We track only free blocks, so we can use payload area
  - The “next” free block could be anywhere
    - We need to store forward/backward pointers, not just sizes
  - Still need boundary tags for coalescing
    - To find adjacent blocks according to memory order

Freeing With Explicit Free Lists

- Insertion policy: where in the free list do you put a newly freed block?
- Unordered
  - LIFO (last-in-first-out) policy
    - Insert freed block at the beginning of the free list
  - FIFO (first-in-first-out) policy
    - Insert freed block at the end 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:

    $addr(prev) < addr(curr) < addr(next)$
  - Con: requires search
  - Pro: studies suggest fragmentation is lower than LIFO/FIFO

Freeing with a LIFO Policy

- Case 1: allocated $\leftrightarrow$ target $\leftrightarrow$ allocated
  - Insert the freed block at the root of the list
- Case 2: allocated $\leftrightarrow$ target $\leftrightarrow$ free
  - Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list
- Case 3: free $\leftrightarrow$ target $\leftrightarrow$ allocated
  - Splice out adjacent successor block, coalesce both memory blocks, and insert the new block at the root of the list
- Case 4: free $\leftrightarrow$ target $\leftrightarrow$ free
  - Splice out adjacent successor block, coalesce all three memory blocks, and insert the new block at the root of the list

An Implementation Trick

- Use circular, doubly linked list
- Support multiple approaches with single data structure
- First-fit versus next-fit
  - Either keep free pointer fixed or move as search list
- LIFO versus FIFO
  - Insert as next block (LIFO) or previous block (FIFO)

Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of *free* blocks instead of *all* blocks (much faster)
  - Slightly more complicated allocate and free because we need to splice blocks in and out of the list
  - Some extra space for the links (two extra words needed for each block)
    - Does this increase internal fragmentation?

Segregated List (Seglist) Allocators

- Each *size class* of blocks has its own free list
  - Example size classes: 16, 32-48, 64-inf
- Often have separate classes for each small size
- For larger sized: one class for each size $[2^i + 1, 2^{i+1}]$

Seglist Allocator

- Given an 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 $m > n$ (that is, first fit)
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list
    - If no block is found, try next larger class
  - Repeat until block is found
- If no block is found:
  - Request additional heap memory from OS (using `sbrk()`)
  - Allocate block of n byte from this new memory
  - Place remainder as single free block in appropriate size class

Seglist Allocator (Continued)

- To free a block:
  - Coalesce and place on appropriate list
- Advantages of seglist allocators versus non-seglist allocators (both with first fit)
  - Higher throughput
    - log time for power-of-two size classes versus linear time
  - Better memory utilization
    - First fit search of segregated free list approximates a best fit search of the entire heap
    - Extreme case: giving each block its own size class is equivalent to best fit

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);
```

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)); // <-- here
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

- Can avoid by using `calloc`

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));
}
```

- Can you spot the bug?

Overwriting Memory

- Off-by-one errors

```
char **p;

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

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

```
char *p;

p = malloc(strlen(s));
strcpy(p,s);
```

Overwriting Memory

- Not checking the max string size

  ```
  char s[8]; // <-- too small
  int i;

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

- Basis for classic buffer overflow attacks

Overwriting Memory

- Misunderstanding pointer arithmetic

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

   while (p && *p != val)
      p += sizeof(int); // <-- here

   return p;
}
```

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--;  // <-- here
   Heapify(binheap, *size, 0);
   return(packet);
}
```

- What gets decremented?
  - Hint: precedence and associativity

Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

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

   return &val;
}
```

Freeing Blocks Multiple Times

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

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

Referencing Freed Blocks

```
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)

```
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

- Debugger: `gdb`
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs
- Data structure consistency checker
  - Runs silently, prints message only on error
  - Use as a probe to zero in on error
- Binary translator: `valgrind`
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block
- `glibc malloc` contains checking code

Implicit Memory Management: Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage; application never has to explicitly free memory
- Common in many dynamic languages
- Variants (“conservative” garbage collectors) exist for C and C++

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
- Must 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 (for example, coercing them to an `int` and then back again)

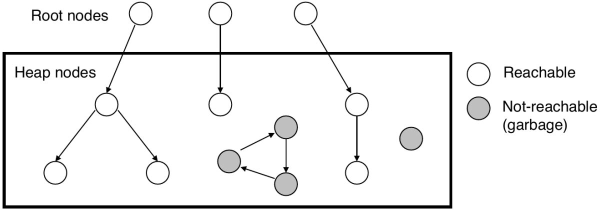
Classical Garbage Collection Algorithms

- 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

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 (for example, registers, locations on the stack, global variables)

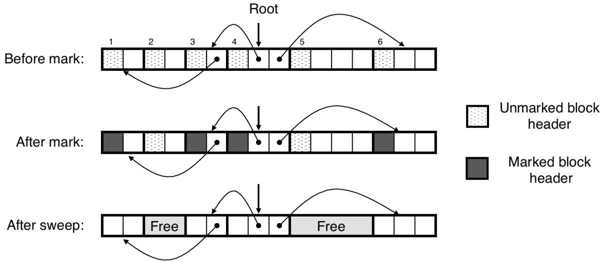
Memory as a Graph



- 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
  - `new(n)`: returns pointer to new block with all locations cleared
  - `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 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 Pseudocode

- Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;          // if not pointer ->
   if (markBitSet(p)) return;       // if already marked
   setMarkBit(p);                   // set the mark bit
   for (i=0; i < length(p); i++)    // for each word in p
       mark(p[i]);                  //  make recursive ca
   return;
}
```

Mark and Sweep Pseudocode

- Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {                  // for entire heap
      if markBitSet(p)                // did we reach this
         clearMarkBit();              //  yes -> so just cl
      else if (allocateBitSet(p)) // never reached: is
         free(p);                     //  yes -> its garbag
      p += length(p+1);               // goto next block
}
```

Conservative Mark and Sweep in C

- A "conservative garbage collector" for C programs
  - `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
- To mark header, need to find the beginning of the block
  - Can use a balanced binary tree to keep track of all allocated blocks (key is start-of-block)
  - Balanced tree pointers can be stored in header (use two additional words)