Dynamic Memory Allocation: Advanced Topics

CSCI 2021: Machine Architecture and Organization

Pen-Chung Yew
Department Computer Science and Engineering
University of Minnesota

Review: Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized blocks, which are either allocated or free
- Types of allocators
  - Explicit allocator: application allocates and frees space
    - e.g., malloc and free in C
  - Implicit allocator: application allocates, but does not free space
    - e.g. garbage collection in Java, ML, and Lisp

Outline

- Garbage collection
- Memory-related perils and pitfalls

Implicit Memory Management: Garbage Collection

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

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

- Common in many dynamic languages:
  - Python, Ruby, Java, Perl, ML, Lisp, Mathematica
- Variants ("conservative" garbage collectors) exist for C and C++
  - However, cannot necessarily 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
- 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 (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:
Memory as a Graph

- We view memory as a directed graph
- Each allocated 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)

![Graph Diagram]

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).

Assumptions For a Simple Implementation

- Application
  - `new (a)`: returns pointer to new block with all locations cleared
  - `read (b, i, v)`: read location `i` of block `b` into register `v`
  - `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 (i)`: 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 each reachable block
  - Sweep: Scan all blocks and free blocks that are not marked

![Mark and Sweep Diagram]

Before mark

After mark

After sweep

Mark and Sweep (cont.)

Mark using depth-first traversal of the memory graph

```c
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++) // call mark on all words
        mark(p[i]); // in the block
    return;
}
```

Sweep using lengths to find next block

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

Conservative Mark & Sweep in C

- A “conservative garbage collector” for C programs
  - `is_ptr (p)`: determines if a word is a pointer by checking if it points to an allocated block of memory
  - But, C pointers can point to the middle of a block

![Conservative Diagram]

So how 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)

Outline

- Garbage collection
- Memory-related perils and pitfalls
Memory-Related Perils and Pitfalls

1. Dereferencing bad pointers
2. Reading uninitialized memory
3. Overwriting memory
4. Referencing nonexistent variables
5. Freeing blocks multiple times
6. Referencing freed blocks
7. Failing to free blocks

C operators – Precedence Rules

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[] () -&gt; .</td>
<td>left to right</td>
</tr>
<tr>
<td>/ %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>^</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^=</td>
<td>right to left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: K&R page 53

C Pointer Declarations: Test Yourself!

int *p
int *p[13]
int *(p[13])
int **p
int *(p[13])
int *f()
int (*f())()
int (*(*f())[13])()
int (*(*x[3])[])[5]

Source: K&R Sec 5.12

1. Dereferencing Bad Pointers
   - The classic scanf bug

```
int val;
...
scanf("%d", &val);
```

2. 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;
}
```
3. Overwriting Memory

- Allocating the (possibly) wrong sized object

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

3. Overwriting Memory (cont.)

- Off-by-one error

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

3. Overwriting Memory (cont.)

- Not checking the max string size

```c
char s[8];
int i;
gets(s); /* reads "123456789" from stdin */
```

- Basis for classic buffer overflow attacks

3. Overwriting Memory (cont.)

- Misunderstanding pointer arithmetic

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

3. Overwriting Memory (cont.)

- Referencing a pointer instead of the object it points to

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

4. Referencing Nonexistent Variables

- Forgetting that local variables disappear when a function returns

```c
int *foo () {
  int val;
  return &val;
}
```
5. Freeing Blocks Multiple Times

- Nasty!

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

6. Referencing Freed Blocks

- Evil!

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

7. Failing to Free Blocks (Memory Leaks)

- Slow, long-term killer!

```c
foo() {
  int *x = malloc(N*sizeof(int));
  ...
  return;
}
```

7. Failing to Free Blocks (Memory Leaks)

- Freeing only part of a data structure

```c
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;
}
```

Memory-Related Perils and Pitfalls

1. Dereferencing bad pointers
2. Reading uninitialized memory
3. Overwriting memory
4. Referencing nonexistent variables
5. Freeing blocks multiple times
6. Referencing freed blocks
7. Failing to free blocks

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
  - `setenv MALLOC_CHECK=3`