6.087 Lecture 11 – January 26, 2010

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Review: C standard library

- I/O functions: `fopen()`, `freopen()`, `fflush()`, `remove()`, `rename()`, `tmpfile()`, `tmpnam()`, `fread()`, `fwrite()`, `fseek()`, `ftell()`, `rewind()`, `clearerr()`, `feof()`, `ferror()`
- Character testing functions: `isalpha()`, `isdigit()`, `isalnum()`, `iscntrl()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`, `isupper()`
- Memory functions: `memcpy()`, `memmove()`, `memcmp()`, `memset()`

Review: C standard library

- Conversion functions: `atoi()`, `atol()`, `atof()`, `strtol()`, `strtoul()`, `strtod()`
- Utility functions: `rand()`, `srand()`, `abort()`, `exit()`, `atexit()`, `system()`, `bsearch()`, `qsort()`
- Diagnostics: `assert()` function, `__FILE__`, `__LINE__` macros

Review: C standard library

- Variable argument lists:
  - Declaration with `...` for variable argument list (may be of any type):
    ```
    int printf (const char * fmt, ...);
    ```
  - Access using data structure `va_list ap`, initialized using `va_start()`, accessed using `va_arg()`, destroyed at end using `va_end()`
- Time functions: `clock()`, `time()`, `difftime()`, `mktime()`, `asctime()`, `localtime()`, `ctime()`, `strftime()`

6.087 Lecture 11 – January 26, 2010

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Dynamic memory allocation

- Memory allocated during runtime
- Request to map memory using `mmap()` function (in `<sys/mman.h>`)
- Virtual memory can be returned to OS using `munmap()`
- Virtual memory either backed by a file/device or by *demand-zero* memory:
  - all bits initialized to zero
  - not stored on disk
  - used for stack, heap, uninitialized (at compile time) globals

Mapping memory

- Mapping memory:

```
void *mmap(void *start, size_t length, int prot,
           int flags, int fd, off_t offset);
```

  - asks OS to map virtual memory of specified length, using specified physical memory (file or demand-zero)
  - `fd` is file descriptor (integer referring to a file, not a file stream) for physical memory (i.e. file) to load into memory
  - for demand-zero, including the heap, use `MMAP_ANON` flag
  - `start` – suggested starting address of mapped memory, usually NULL

- Unmap memory:

```
int munmap(void *start, size_t length);
```

The heap

- Heap – private section of virtual memory (demand-zero) used for dynamic allocation
- Starts empty, zero-sized
- `brk` – OS pointer to top of heap, moves upwards as heap grows
- To resize heap, can use `sbrk()` function:

```c
void *sbrk(int inc); /* returns old value of brk_ptr */
```

- Functions like `malloc()` and `new` (in C++) manage heap, mapping memory as needed
- Dynamic memory allocators divide heap into blocks

Requirements

- Must be able to allocate, free memory in any order
- Auxiliary data structure must be on heap
- Allocated memory cannot be moved
- Attempt to minimize fragmentation

Fragmentation

- Two types – internal and external
- Internal – block size larger than allocated variable in block
- External – free blocks spread out on heap
- Minimize external fragmentation by preferring fewer larger free blocks

Design choices

- Data structure to track blocks
- Algorithm for positioning a new allocation
- Splitting/joining free blocks

Tracking blocks

- Implicit free list: no data structure required
- Explicit free list: heap divided into fixed-size blocks; maintain a linked list of free blocks
  - allocating memory: remove allocated block from list
  - freeing memory: add block back to free list
- Linked list iteration in linear time
- Segregated free list: multiple linked lists for blocks of different sizes
- Explicit lists stored within blocks (pointers in payload section of free blocks)

Block structures

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 10.37, Format of a simple heap block.

Block structures

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 10.50, Format of heap blocks that use doubly-linked free lists.

Positioning allocations

- Block must be large enough for allocation
- First fit: start at beginning of list, use first block
- Next fit: start at end of last search, use next block
- Best fit: examines entire free list, uses smallest block
- First fit and next fit can fragment beginning of heap, but relatively fast
- Best fit can have best memory utilization, but at cost of examining entire list

Splitting and joining blocks

- At allocation, can use entire free block, or part of it, splitting the block in two
- Splitting reduces internal fragmentation, but more complicated to implement
- Similarly, can join adjacent free blocks during (or after) freeing to reduce external fragmentation
- To join (coalesce) blocks, need to know address of adjacent blocks
- Footer with pointer to head of block – enable successive block to find address of previous block

A simple memory allocator

- Code in *Computer Systems: A Programmer's Perspective*
- Payload 8 byte alignment; 16 byte minimum block size
- Implicit free list
- Coalescence with boundary tags; only split if remaining block space ≥ 16 bytes

Initialization

1. Allocate 16 bytes for padding, prologue, epilogue
2. Insert 4 byte padding and prologue block (header + footer only, no payload) at beginning
3. Add an epilogue block (header only, no payload)
4. Insert a new free chunk (extend the heap)

Allocating data

1. Compute total block size (header+payload+footer)
2. Locate free block large enough to hold data (using first or next fit for speed)
3. If block found, add data to block and split if padding ≥ 16 bytes
4. Otherwise, insert a new free chunk (extending the heap), and add data to that
5. If could not add large enough free chunk, out of memory

Freeing data

1. Mark block as free (bit flag in header/footer)
2. If previous block free, coalesce with previous block (update size of previous)
3. If next block free, coalesce with next block (update size)

Explicit free list

- Maintain pointer to head, tail of free list (not in address order)
- When freeing, add free block to end of list; set pointer to next, previous block in free list at beginning of payload section of block
- When allocating, iterate through free list, remove from list when allocating block
- For segregated free lists, allocator maintains array of lists for different sized free blocks

`malloc()` for the real world

- Used in GNU libc version of `malloc()`
- Details have changed, but nice general discussion can be found at
  `http://g.oswego.edu/dl/html/malloc.html`
- Chunks implemented as in segregated free list, with pointers to previous/next chunks in free list in payload of free blocks
- Lists segregated into bins according to size; bin sizes spaced logarithmically
- Placement done in best-fit order
- Deferred coalescing and splitting performed to minimize overhead

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Using `malloc()`

- Minimize overhead – use fewer, larger allocations
- Minimize fragmentation – reuse memory allocations as much as possible
- Growing memory – using `realloc()` can reduce fragmentation
- Repeated allocation and freeing of variables can lead to poor performance from unnecessary splitting/coalescing (depending on implementation of `malloc()`)

Using `valgrind` to detect memory leaks

- A simple tutorial: `http://cs.ecs.baylor.edu/~donahoo/tools/valgrind/`
- `valgrind` program provides several performance tools, including `memcheck`:

  ```
  athena%[1] valgrind --tool=memcheck
  --leak-check=yes program.o
  ```

- `memcheck` runs program using virtual machine and tracks memory leaks
- Does not trigger on out-of-bounds index errors for arrays on the stack

[1] Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

Other `valgrind` tools

- Can use to profile code to measure memory usage, identify execution bottlenecks
- `valgrind` tools (use name in `-tool=` flag):
  - `cachegrind` – counts cache misses for each line of code
  - `callgrind` – counts function calls and costs in program
  - `massif` – tracks overall heap usage

- Review
- Dynamic Memory Allocation
  - Designing the `malloc()` Function
  - A Simple Implementation of `malloc()`
  - A Real-World Implementation of `malloc()`
- Using `malloc()`
  - Using `valgrind`
- Garbage Collection

Garbage collection

- C implements no garbage collector
- Memory not freed remains in virtual memory until program terminates
- Other languages like Java implement garbage collectors to free unreferenced memory
- When is memory unreferenced?

Garbage collection

- C implements no garbage collector
- Memory not freed remains in virtual memory until program terminates
- Other languages like Java implement garbage collectors to free unreferenced memory
- When is memory unreferenced?
  - Pointer(s) to memory no longer exist
  - Tricky when pointers on heap or references are circular (think of circular linked lists)
  - Pointers can be masked as data in memory; garbage collector may free data that is still referenced (or not free unreferenced data)

Garbage collection and memory allocation

- Program relies on garbage collector to free memory
- Garbage collector calls `free()`
- `malloc()` may call garbage collector if memory allocation above a threshold

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 10.52, Integrating a conservative garbage collector and a C malloc package.

Mark and sweep garbage collector

- Simple tracing garbage collector
- Starts with list of known in-use memory (e.g. the stack)
- Mark: trace all pointers, marking data on the heap as it goes
- Sweep: traverse entire heap, freeing unmarked data
- Requires two complete traversals of memory, takes a lot of time
- Implementation available at `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`

Mark and sweep garbage collector

Figure removed due to copyright restrictions. Please see
http://csapp.cs.cmu.edu/public/1e/public/figures.html,
Figure 10.51, A garbage collector's view of memory as a directed graph.

Mark and sweep garbage collector

Figure removed due to copyright restrictions. Please see http://csapp.cs.cmu.edu/public/1e/public/figures.html, Figure 10.54, Mark and sweep example.

Copying garbage collector

- Uses a duplicate heap; copies live objects during traversal to the duplicate heap (the *to-space*)
- Updates pointers to point to new object locations in duplicate heap
- After copying phase, entire old heap (the *from-space*) is freed
- Code can only use half the heap

Cheney's (not Dick's) algorithm

- Method for copying garbage collector using breadth-first-search of memory graph
- Start with empty to-space
- Examine stack; move pointers to to-space and update pointers to to-space references
- Items in from-space replaced with pointers to copy in to-space
- Starting at beginning of to-space, iterate through memory, doing the same as pointers are encountered
- Can accomplish in one pass

Summary

Topics covered:

- Dynamic memory allocation
    - the heap
    - designing a memory allocator
    - a real world allocator
- Using `malloc()`
- Using `valgrind`
- Garbage collection
    - mark-and-sweep collector
    - copying collector

6.087 Practical Programming in C
January (IAP) 2010