

Memory Management

DSA C++

- general ideas behind basic malloc, realloc, free/new & delete
- different levels of memory management (hardware MMU, run time, garbage collection)
- external and internal fragmentation management

Sequential-fit

- for efficiency, use doubly linked list inside a block of memory, also maintaining a status field and a size field
- approaches:
 - first-fit: scan through the list and allocate the first block of memory large enough to meet the request
 - best-fit: scans through to find the block that is closest in fit to the requested size
 - worst-fit: use largest block in list so that the remaining portion is large enough to be used in future requests
 - next-fit: use the next available block that is sufficiently large
- analysis:
 - first-fit: most efficient
 - next-fit: comparable in speed but causes more external fragmentation
 - best-fit: causes the most fragmentation
 - worst-fit: avoids fragmentation in some ways
- fragmentation may be more dependent on the implementation than the method

Nonsequential-fit

- sequential-fit methods may become inefficient for large memory
- approach:
 - divide memory into an arbitrary number of lists, each holding blocks of the same size
 - larger blocks are split and the remainder is added to other lists or new lists are created if necessary
 - list sizes can be ordered into trees (Cartesian trees)
 - variants of first-fit and best-fit can be applied (i.e leftmost-fit and better-fit)
- approach:
 - programs have a limited number of sizes requested
 - use the above list approach with the most popular block sizes
 - known as the adaptive exact-fit approach

- need to handle memory fragmentation:
 - occasional compaction
 - can also liquidate size list and rebuild

First-fit method (from TAOC 1)

- creates a linked list with the head pointing to the first available block
- each node has two fields for the related address
 - size of the memory contained at address P (i.e. number of words)
 - a link to the address of the next available block
- the last pointer is null
- algorithm:
 - overview: searches for an reserves the next available block or reports failure
 - variables: available (first available block), p, q
 - assumed preconditions:

```
link(p) returns a pointer to the next available block
available == link(location(available))
```

1. initialize:

```
q = location(available)
```

2. check for end of list:

```
p = link(q)
if (p == nullptr) { return or throw; }
```

3. check for adequate size:

```
if (size(p) >= requested_size) { continue; }
else { q = p; goto 2; }
```

4. reserve the size (and return the location -> edit)

```
k = size(p) - requested_size;
if (k == 0) { link(q) = link(p); return p; }
else { size(p) = k; return p + k; }
```

- NOTE: the algorithm terminates successfully on 4, reserving an area of length n beginning at location $p + k$
- improvement in performance: use the next fit method and rearrange links to maintain increasing order

Liberation with sorted list (from TAOC 1)

- follows from the First-fit method above but adds the assumption that if $\text{link}(p) \neq \text{nullptr}$ then $\text{link}(p) > p$ (i.e. the available list is sorted by memory location)
- adds a block of n consecutive cells beginning at p_0 to the available list with the assumption that none of the n cells is already available
- algorithm:
 - initialize

```
q = location(available)
```

2. advance p

```
p = link(q)
if (p == nullptr || p > p0) { goto 3; }
else { q = p; goto 2; }
```

3. check upper bound

```
if (p0 + n == p && p != nullptr) { n = n + size(p); link(p0) = link(p); }
else { link(p0) = p; }
```

4. check lower bound

```
// assumptions
size(location(available)) = 0;
if (q == location(available)) { throw; }

// otherwise
if (q + size(q) == p0) { size(q) = size(q) + n; link(q) = link(p0); }
else { link(q) = p0; size(p0) = n; }
```

Liberation with boundary tags (from TAOC 1)

- layout:

first word	second word	size - 2 words	last word
use flag size		block ...	

- assumptions:

- memory layout above, doubly linked (possibly in second and last word)
- available is the start of the doubly linked list
- places blocks of locations starting with p0 into the available list
- for the pool storage, locations m0 to m1 inclusive, assumes (tag meaning use flag, true meaning in use or reserved)

```
tag(m0 - 1) == tag(m1 + 1) == true;
```

- algorithm:

- check lower bound

```
if (tag(0 - 1)) { goto 3; }
```

2. delete lower area

```
p = p0 - size(p0 - 1);
p1 = link(p);
p2 = link(p + 1);
link(p1 + 1) = p2;
link(p2) = p1;
size(p) = size(p) + size(p0);
p0 = p;
```

3. check upper bound

```
p = p0 + size(p0);
if (tag(p)) { goto 5; }
```

4. delete upper area

```
p1 = link(p);
p2 = link(p + 1);
link(p1 + 1) = p2;
link(p2) = p1;
size(p0) = size(p0) + size(p);
p = p + size(p);
```

5. add to `available` list

```
size(p - 1) = size(p0);
link(p0) = available;
link(p0 + 1); = location(available);
link(available + 1) = p0;
available = p0;
tag(p0) = tag(p - 1) = false;
```

Basic memory allocation (from Modern Compiler Design)

- blocks: memory fragments released to user
- chunks: memory fragments controlled by allocator
- heap carves memory into chunks with the last byte of one chunk preceding the first byte of the following chunk

chunk										
			block							

size		use flag		block		size		use flag		block

size indicates size of chunk

- additional considerations - align block start on optimal alignment boundaries for a given data type -> this requires padding the size and flag segment or setting them back to ensure the block start location
- generalized algorithm

```
void* first_chunk = get_start_of_avail_mem();
void* past_end = first_chunk + size_of_avail_mem();

first_chunk->size = size_of_avail_mem();
first_chunk->free = true;

void* malloc(size_t size) {
    void* p = ptr_to_free_block_of_size(size);
    if (p != nullptr) { return p; }

    coalesce_free_chunks();
    void* p = ptr_to_free_block_of_size(size);
    if (p != nullptr) { return p; }

    // throws if fails
    solve_out_of_mem_cond(size);
```

```

        return malloc(size); // retry
    }

void free(void* p) {
    void *chunk = p - (sizeof(size_t) + sizeof(bool));
    chunk->free = true;
}

```

- compiler implementations:
 - linked lists
 - extensible arrays

Buddy Systems

- little external fragmentation and allows for compaction with little overhead
- generalized as a memory pool
- many different approaches
- can be combined with slab allocation to reduce fragmentation

Description (from TAOC 1)

- keep separate lists of available blocks of each size 2^k for $0 \leq k \leq m$
- entire pool consists of 2^m words with addresses 0 through $2^m - 1$
- starts with entire pool available
- when a block of memory is requested, if nothing of that size is available, a larger block is split in two to fulfill the request
- a split block forms two "buddies" and are rejoined when possible
- defined by fact that if the address and size of the first block is known, then the address and size of the second block is also known
- given the address of a block, x , with size 2^k

$$\text{buddy sub } k(x) = \{ x + 2^k, \text{ for } x \bmod 2^{(k+1)} == 0; \\ x - 2^k, \text{ for } x \bmod 2^{(k+1)} == 2^k; \}$$

- makes use of a one bit (or boolean) tag field in each block
- algorithm (reservation):
 - i. find block

```

j = min of range(k, m) if available_f[j] != location(available[j]); // inclusive r
// assumes available is sized 2^j and is not empty
// throw if no such j exists

```

2. remove from list

```

l = available_b[j];
p = link_b(l);
available_b[j] = p;
link_f(p) = location(available[j]);
tag(l) = 0;

```

3. split required if necessary

```

if (j == k) { return address l; }

```

4. split

```

--j;
p = l + 2^j;
tag(p) = 1;
kval(p) = j;
link_f(p) = link_b(p) = location(available[j]);
available_f[j] = available_b[j] = p;
// splits a large block and registers the unused half in the available[j] list whi
goto 3;

```

- algorithm (liberation):
 - i. check if buddy is available

```

p = buddy sub k(l);
if (k == m || tag(p) == 0 || (tag(p) == 1 and kval(p) != k) { goto 3; }

```

2. combine with buddy

```

link_f(link_b(p)) = link_f(p);
link_b(link_f(p)) = link_b(p);
// removes block p from available[k] list
k = k + 1;
if (p < l) { l = p; goto 1; }

```

3. add l to `available` list

```

tag(l) = 1;
p = available_f[k];
link_f(l) = p;
link_b(p) = l;
kval(l) = k;

```

```
link_b(l) = location(available[k])
available_f[k] = l
// adds l to the available[k] list
```

Comparison of methods

- if first-fit and liberation with sorted list are used continuously, when the system tends to equilibrium with n reserved blocks on average, each equally likely to be the next freed block, and k in first-fit assuming non-zero values with probability p , then avg. number of available blocks tends to $\sim 1/2p \cdot n$
- using MIX/MMIX both the boundary tag system and the buddy system are fast for reservation and liberation, but the buddy system is slightly faster
- for known block size distributions, with equally likelihood that any block will be the next freed, the "distributed-fit method" results in much better memory utilization
 - partition memory into $n + n^{1/2} \cdot \lg(n)$
 - n = desired max number of blocks
 - each slot has a fixed size, different slots may have a different size, and any given slot has fixed boundaries and will be empty or contain a single allocated block

Caching (from wikipedia https://en.wikipedia.org/wiki/Cache_replacement_policies)

Belady's/clairvoyant algorithm

- discard information that will not be needed for the longest time in the future

FIFO

- removes element from the cache that has been in the cache for the longest time
- can use deque or queue

LIFO

- removes element from the cache that has been in the cache for the shortest time

- can use vector, stack, or deque

LRU (Least recently used)

- family of caching algorithms
 - LRU-K
 - ARC
 - MQ
 - 2Q
 - LRU-2
 - LRFU
 - LIRS
- involves keeping an age/access bit which is incremented on addition to the cache and incremented on cache access
- when a cache miss occurs, the cache item with the lowest age/access count is evicted and the element causing the cache miss is added

MRU (Most recently used)

- like LRU but discards the item with the highest age/access bit
- useful for repeated looping over an input in a reference pattern, random access patterns, and repeated scans over large data sets (aka cyclic access patterns)
- i.e. most useful when older items are more likely to be accessed

LFU (Least-frequently used)

- much like LRU but only tracks usage and removes the least used on a cache miss

Additional

- Pseudo-LRU (PLRU)
- Random replacement (RR)
- Segmented LRU (SLRU)
- LFU with Dynamic Aging (LFUDA)
- Low Inter-reference Recency Set (LIRS)
- Adaptive Replacement Cache (ARC)
- Clock with Adaptive Replacement (CAR)
- Multi Queue (MQ) caching algorithm | Multi Queue (MQ)

- Pannier: Container-based caching algorithm for compound objects

Garbage collection (wiki and compiler)

- reference counting: uses a usage count to determine when chunks are no longer needed -> requires concept of strong and weak references to avoid retain cycles
- mark and scan (mark and sweep): traverses chunk graphs and identifies reachable chunks, removing unreachable chunks
- two-space copying: maintains a from-space and a to-space, moving reachable chunks to the to-space (efficient but wastes 1/2 memory)
- higher level categorization:
 - tracing: what is commonly referred to as garbage collection. Generally, but not limited to, determining reachable objects and removing unreachable objects
 - reference counting: see above
 - escape analysis: method used to convert heap allocations to stack allocations to reduce garbage collector work -> generally done at compile time
- three varieties:
 - one-shot: runs through to completion
 - on-the-fly/incremental: run on each request for memory, mostly when more memory is needed or improvements to memory fragmentation/chunk structure is needed
 - concurrent: garbage collector runs on a separate processor

Mark and sweep

- naive:
 - each object has a use flag (always cleared except during collection)
 - traverses entire object tree (graph) from root and marks each pointed to object as in use
 - scans all memory from start to finish and frees any objects not marked as in-use
 - requires an entire scan of memory (some twice) which is a problem in paged memory
- tri-coloring:
 - white: candidate for recycling
 - black: no outgoing references but reachable from roots (not candidates)
 - gray: all objects reachable from root but yet to be scanned
 - procedure:
 - move object from gray set to black set
 - move each white object referenced by gray object to gray set
 - repeat last until gray set is empty
 - remove all remaining white objects
 - has the advantage of being able to be performed on-the-fly or incrementally

- moving vs non-moving: may use two areas of memory and move reachable objects to separate area (aka compacting vs. non-compacting)
- copying vs. mark-and-sweep vs. mark-and-don't-sweep
- generational
- stop-the-world vs. incremental vs. concurrent
- precise vs. conservative and internal pointers

Cheney's algorithm

- stop and copy method of tracing garbage collection with a heap divided into two halves
- reclaims by:
 - if the object has not yet been moved to the to-space creates copy in to-space, replaces from-space version with a forwarding pointer to the to-space copy. Updates the object reference to refer to the new version in to-space.
 - if the object has already been moved to the to-space, simply update the reference from the forwarding pointer in from-space.

Generational garbage collector

- see above under high level categorization and tracing

Mark-compact algorithm

- uses mark-and-sweep and Cheney's copying algorithm to move reachable objects to another memory location (i.e. the beginning of the heap)
- used by GHC

Semi-space collector

- copying method mentioned above using from-space and to-space
- performs collection when to-space does not have adequate space to fulfill a request
- allocation costs are extremely low
- requires twice as much memory
- NOTE: Knuth and Compiler Techniques have more discussions in this area
- memory management
 - sequential-fit methods
 - nonsequential-fit methods
 - garbage collection
 - mark-and-sweep

- copying methods
- incremental
- generational
- caching