Advanced C Programming

Memory Management II
(malloc, free, alloca, obstacks, garbage collection)

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

16.12.2008

Contents

Problems of Memory Allocation

Fragmentation

- Not being able to reuse free memory
- Free memory is split up in many small pieces
- Cannot reuse them for large-piece requests
- Primary objective of today's allocators is to avoid fragmentation

Locality

- Temporal and spacial locality go along with each other
- Memory accesses near in time are also near in space
- Try to serve timely near requests with memory in the same region
  ☞ Less paging
- Memory allocation locality not that important for associative caches
  ☞ Enabling locality by the programmer more important

Practical Considerations (see [Lea])

A good memory allocator needs to balance a number of goals:

Minimizing Space

- The allocator should not waste space
- Obtain as little memory from the system as possible
- Minimize fragmentation

Minimizing Time

- `malloc`, `free` and `realloc` should be as fast as possible in the average case

Maximizing Tunability

- Configure optional features (statistics info, debugging, ...)

Maximizing Locality

- Allocate chunks of memory that are typically used together near each other
- Helps minimize page and cache misses during program execution

Minimizing Anomalies

- Perform well across wide range of real loads

Approaches

- Allocate and Free
  - Allocating and freeing done by the programmer
  - Bug-prone: Can access memory after being freed
  - Potentially efficient: Programmer should know when to free what
- Garbage Collection
  - User allocates
  - System automatically frees dead chunks
  - Less bug-prone
  - Potentially inefficient:
    Overhead of the collection, many dead chunks
- Region-based approaches
  - User allocates chunks inside a region
  - Only the region can be freed
  - Efficiency of allocate and free
  - Slightly less bug-prone
  - many dead chunks

Allocation on the stack

- If you know that the allocated memory will be only used during life time of a function
- Allocate the memory in the stack frame of the function
- Allocation costs only increment of stack pointer
- Freeing is “free” because stack pointer is restored at function exit
- Don’t do it for recursive functions (stack might grow too large)

```
void foo(int n) {
    int *arr = alloca(n * sizeof(*arr));
    ...
}
```

- Only do this if you do not statically know the size of the memory to allocate
- `alloca` is strongly machine and compiler dependent and not POSIX!
  ☞ Only use if absolutely necessary
- In C99, use VLAs instead (unfortunately not in C++)

Malloc and free

In every execution of the program, all allocated memory should be freed

- Make it proper ☞ make it more bug-free
- Never waste if you don't need to
- You might make a library out of your program
- People using that library will assume proper memory management

Purpose of `malloc`, `free`

- Get memory for the process from OS (`mmap`, `sbrk`, ...)
- Manage freed memory for re-utilization

Getting Memory from the OS (UNIX)

Unices usually provide two syscalls to enlarge the memory of a process:

- `brk`
  - Move the end of the uninitialized data segment
  - At the start of the program, the break is directly behind the uninitialized data segment of the loaded binary
  - Moving the break adds memory to the process
  - `malloc` has to set the break as tightly as possible
    ☞ deal with fragmentation
  - Reuse unused memory below the break
  - `brk` is fast

- `mmap`
  - Map in pages into a process' address space
  - Finest granularity: size of a page (usually 4K)
  - More overhead in the kernel than `brk`
  - Used by `malloc` only for large requests ($>$ 1M)
    ☞ Reduces fragmentation: pages can be released independently from each other

Contents

The Doug Lea Allocator (DL malloc)

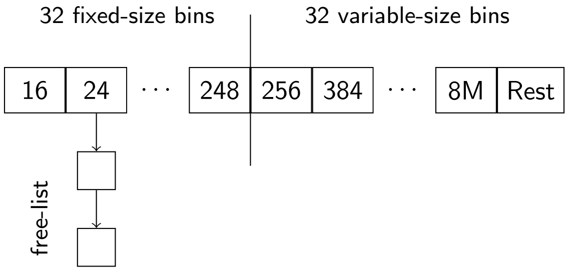
- Base of glibc `malloc`
- One of the most efficient allocators
- Very fast due to tuned implementation

- Uses a best-fit strategy:
  ☞ Re-use the free chunk with the smallest waste
- Coalesces chunks upon free
  ☞ Reduce fragmentation
- Uses binning to find free chunks fast

- Smallest allocatable chunk:
  - 32-bit system: 8 bytes + 8 bytes bookkeeping
  - 64-bit system: 16 bytes + 16 bytes bookkeeping

Binning

- Goal: Find the best-fitting free chunk fast
- Solution: Keep bins of free-lists/trees
- Requests for small memory occur often
  - Split bins into two parts
  - 32 exact-size bins for everything up to 256 bytes
  - 32 logarithmic scaled bins up to $2^{\text{pointer size}}$



Searching the best-fitting Chunk

Small Requests $<$ 256 bytes

- Check if there is a free chunk in the corresponding exact-size bin
- If not, look into the next larger exact-size bin and check there
- If that bin had no chunk too, check the designated victim (dv) chunk
- If the `dv` chunk was not sufficiently large
  - search the smallest available small-size chunk
  - split off a chunk of needed size
  - make the rest the designated victim chunk
- If no suitable small-size chunk was found
  - split off a piece of a large-size chunk
  - make the remainder the new dv chunk
- Else, get memory from the system

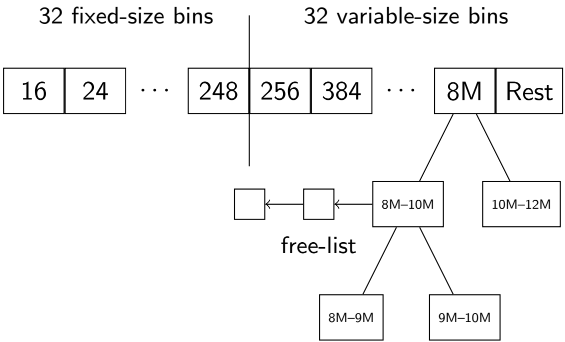
Remark

Using the `dv` chunk provides some locality as unserved requests get memory next to each other

Searching the best-fitting Chunk

Large Requests $\geq$ 256 bytes

- Non-exact bins organize the chunks as binary search trees
- Two equally spaced bins for each power of two
- Every tree node holds a list of chunks of the same size
- Tree is traversed by inspecting the bits in size (from more significant to less significant)
- Everything above 12M goes into the last bin (usually very rare)

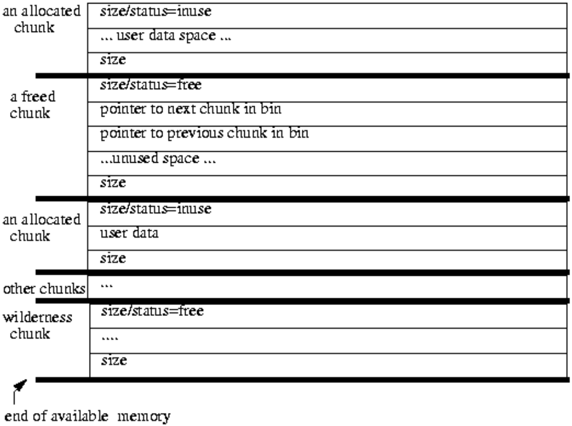


What happens on a `free`?

- Coalesce chunk to free with surrounding free chunks
- Treat special cases if one of the surrounding chunks is `dv`, mmap'ed, the wilderness chunk
- Reinsert the (potentially coalesced) chunk into the free list/tree of the according bin
- Coalescing very fast due to "boundary tag trick":
  Put the size of a free chunk its beginning and its end

Chunk Coalescing

- If a chunk is freed it is immediately coalesced with free blocks around it (if there are any)
- Free blocks are always as large as possible
- Avoid fragmentation
- Faster lookup because there are fewer blocks
- Invariant: The surrounding chunks of a chunk are always occupied



Contents

Region-based Memory Allocation

- Get a large chunk of memory
- Allocate small pieces out of it
- Can free only the whole region
- Not particular pieces within the region

Advantages:

- Fast allocation/de-allocation possible
- Engineering
  - Can free many things at once
  - Very good for phase-local data
    (data that is only used in a certain phase in the program)
  - Think about large data structures: graphs, trees, etc.
    Do not need to traverse to free each node

Disadvantages:

- Potential large waste of memory

Obstacks (Object Stacks)

Introduction

- Region-based memory allocation in the GNU C library
- Memory is organized as a stack:
  - Allocation/freeing sets the stack mark
  - Cannot free single chunks inside the stack
- Can be used to “grow” an object:
  Size of the object is not yet known at allocation site
- Works on top of `malloc`

Allocation / Deallocation

```
void test(int n) {
    struct obstack obst;
    obstack_init(&obst);

    /* Allocate memory for a string of length n-1 */
    char *str = obstack_alloc(&obst, n * sizeof(str[0]));

    /* Allocate an array for n nodes */
    node_t **nodes = obstack_alloc(&obst, n * sizeof(nodes[0]));

    /* Store the current mark of the obstack */
    void *mark = obstack_base(&obst);

    /* Allocate the nodes */
    for (i = 0; i < n; i++)
        nodes[i] = obstack_alloc(&obst, sizeof(node[0]));

    /* All the marks are gone */
    obstack_free(&obst, mark);

    /* Everything has gone */
    obstack_free(&obst, NULL);
}
```

Growing an obstack

- Sometimes you do not know the size of the data in advance (e.g. reading from a file)
- Usually, you to `realloc` and copy
- obstacks do that for you
- Cannot reference data in growing object while growing addresses might change because grow might copy the chunk
- Call `obstack_finish` when you finished growing Get a pointer to the grown object back

```
int *read_ints(struct obstack *obst, FILE *f) {
    while (!feof(f)) {
        int x, res;
        res = fscanf(f, "%d", &x);
        if (res == 1)
            obstack_int_grow(obst, x);
        else
            break;
    }
    return obstack_finish(obst);
}
```

Contents

Garbage Collection

- Garbage collection is the automatic reclamation of memory that is no longer in use
- "Write mallocs without frees"
- Basic principle:
  - At each moment we have a set of roots into the heap: pointers in registers, on the stack, in global variables
  - These point to objects in the heap which in turn point to other objects
  - All objects and pointers form a graph
  - Perform a search on the graph starting from the roots
  - All non-reachable objects can no longer be referenced
  - Their memory can thus be reclaimed
- Major problems for C/C++:
  - Get all the roots
  - Determine if a word is a pointer to allocated memory

The Boehm-Demers-Weiser Collector [Boehm]

- Compiler-independent implementation of a C/C++ garbage collector
- Can co-exist with `malloc` ☞ keeps its own area of memory
- Simple to use: Exchange `malloc` with `GC_malloc`
- Collector runs in allocating thread: collects upon allocation

- Uses mark-sweep allocation:
  1. Mark all objects reachable from roots
  2. Repeatedly mark all objects reachable from newly marked objects
  3. Sweep: Reuse unmarked memory ☞ put into free lists

- Allocation for large and small objects is different:
  - Allocator for small objects gets a "page" from the large allocator
  - Has separate free lists for small object sizes
  - Invariant: All objects in a page have the same size

Getting the Roots

- Roots are in:
  - Processor's registers
  - Values on the stack
  - Global variables (also dynamically loaded libraries!)

- Awkwardly system dependent
- Need to be able to write registers to the stack (`setjmp`)
- Need to know the bottom of the stack
- Quote from Boehm's slides: "You don't wanna know"

Checking for Pointers

Is `0x0001a65a` a pointer to an allocated object?

- Compare word against upper and lower boundaries of the heap
- Check if potential pointer points to a heap page that is allocated
- Potentially, the pointer points in the middle of the object
  ☞ fixup required to get object start address

- Method is conservative:
- Words might be classified although they are none
- memory that is no longer in use might not be freed
- However: Values used in pointers seldom occur as integers

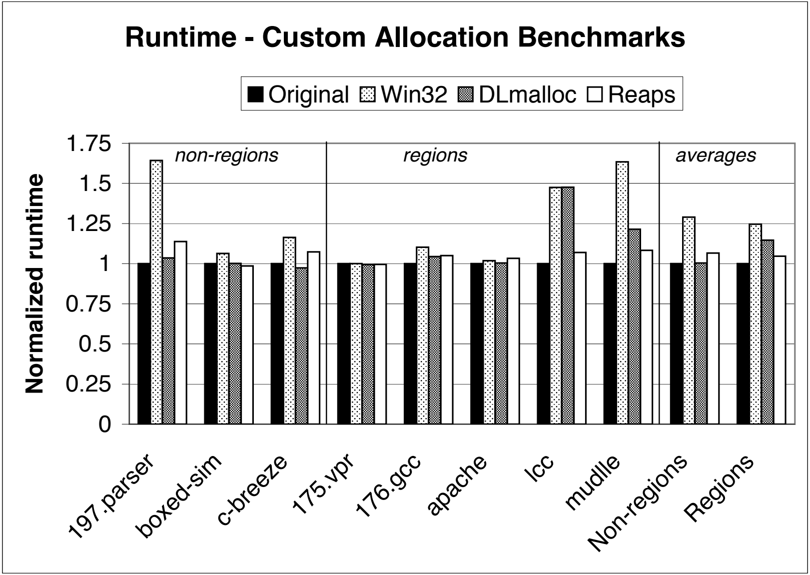
A Critique of Custom Memory Allocation

- Berger et al. [Berger 2002] compared custom allocation to the Windows `malloc` and DL malloc
- Programs from the SPEC2000 benchmark suite and others
- Some having custom allocators, some using general-purpose `malloc`/`free`
- Programs with GP-allocation spend 3% in memory allocator
- Programs with custom allocation spend 16% in memory allocator
- Almost all programs do not run faster with custom allocation compared to DL malloc
- Only programs using region-based allocators are still faster
- DL malloc eliminates most performance advantages by custom allocators

Conclusion

- Use region-based allocation (obstacks) for engineering advantages and fast alloc/free
- When regions are not suitable, use DL malloc

A Critique of Custom Memory Allocation



References

Doug Lea
A memory allocator
http://g.oswego.edu/dl/html/malloc.html

Emery Berger, Benjamin Zorn, and Kathryn McKinley
Reconsidering Custom Memory Allocation, OOPSLA'02
http://www.cs.umass.edu/~emery/pubs/berger-oopsla2002.pdf

Hans-J. Boehm
Conservative GC Algorithmic Overview
http://www.hpl.hp.com/personal/Hans_Boehm/gc/gcdescr.html

Further Reading

Paul Wilson
Uniprocessor Garbage Collection Techniques
ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles
Dynamic Storage Allocation: A Survey and Critical Review
http://www.cs.northwestern.edu/ pdinda/ics-s05/doc/dsa.pdf

Hans-J. Boehm
The “Boehm-Demers-Weiser” Conservative Garbage Collector, Tutorial ISMM’04
http://www.hpl.hp.com/personal/Hans_Boehm/gc/04tutorial.pdf