What is the common most-accessed subroutine of all executed C&C++ programs?

Transitively also of programs in Java, C# and .NETs, Python, Rust, JavaScript, LISPs, Haskell, MLs, Golang, R, PHP, Ruby, shells, …
MOTIVATION

What is the common most-accessed subroutine of all executed C&C++ programs?

Transitively also of programs in Java, C# and .NETs, Python, Rust, JavaScript, LISPs, Haskell, MLs, Golang, R, PHP, Ruby, shells, ...

malloc()
(or a matching variant thereof)
CUSTOM RESTRICTED MEMORY ALLOCATOR

Assignment #1

Advanced C++ course, KSI MFF UK
Memory management is a grave concern for implementation of programming languages and low-level programs.

- Inner workings of the allocator are usually considered black magic.
Memory management is a grave concern for implementation of programming languages and low-level programs.

- Inner workings of the allocator are usually considered black magic.
- Studying the inner workings of commonly used allocators confirms the black magic.
• Show that it is not that hard
  (on a slightly simplified, non-general case)
• Show that it is not that hard (on a slightly simplified, non-general case)
• Practice the Allocator named requirement of C++ classes
• Practice some common data structures used for implementing allocators
• Practice working with raw memory
Some of the main concerns of memory allocator design
Program interface  user asks for *non-overlapping memory blocks*

**OS interface**  the allocator can obtain potentially infinite amount of memory using syscalls

**Middle layer problems**  Mixed use of available memory
- control structures required for ensuring that the memory does not overlap
- memory blocks for the user

**Concerns**
- Consume the smallest possible amount of OS resources
- Once allocated, user blocks can not be moved
- Allocation/deallocation must be fast
- Syscalls are slow
• glibc malloc internals:  
  https://sourceware.org/glibc/wiki/MallocInternals

• Doug Lea’s malloc (used before 2000):  
  http://g.oswego.edu/dl/html/malloc.html

• jemalloc: http://jemalloc.net/

• SLAB/SLUB allocators in Linux kernel

• ... ... ...

Common implementation concerns: Minimize space, time, and anomalies; maximize locality, allow tuning.
Bitmaps are the simplest (and reasonably powerful) way to store allocation information.

- Describing vacancy in $n$ blocks of memory takes $n$ bits.
- For $n$ blocks of $b$ bits, we need $n \cdot (1 + b)$ bits.
Operations:

Find a free block  Scan the vacancy bits, return the position of the first clear bit
Possible size vs. speed tradeoffs:
  • remember a position for starting the scan
  • remember total vacancy

Allocate a block  Set the bit
Deallocate a block  Clear the bit
Operations:

**Find a free block**  Scan the vacancy bits, return the position of the first clear bit

Possible size vs. speed tradeoffs:
- remember a position for starting the scan
- remember total vacancy

**Allocate a block**  Set the bit

**Deallocate a block**  Clear the bit

What to do with multi-block allocations?
WHAT TO DO WITH MULTI-BLOCK ALLOCATIONS?

Naive approach: Store extra information about allocation size in the bitmap.
WHAT TO DO WITH MULTI-BLOCK ALLOCATIONS?

Naive approach: Store extra information about allocation size in the bitmap.

Problems:

• Does not mix tiny vs. big blocks very well
• Eats extra memory for size annotation
WHAT TO DO WITH MULTI-BLOCK ALLOCATIONS?

Naive approach: Store extra information about allocation size in the bitmap.

Problems:

- Does not mix tiny vs. big blocks very well
- Eats extra memory for size annotation

Usual solution: Separate the bitmaps for small and big allocations.
To aid separation, memory is usually divided into chunks.

- Chunks form a double-linked list in the whole heap.
- Chunks contain
  - pointers to other chunks (the list may be circular)
  - memory for block allocation
    - bitmap
    - just a single block
  - extra helpful information
    - was the chunk mmap-ed or does it reside on heap?
    - how big is the chunk
    - what is the size of bitmap blocks
- Various alternative designs exist
  (layered/multi-arena chunks, interval trees, …)
Initialize Find heap dimensions, store pointer to heap.

Create a chunk of sufficient size

1. Run through the linked list and find a free piece of space between adjacent nodes
2. If there is no free space, ask OS for more
3. Modify the linked list.

Remove a free chunk

1. Modify the linked list to skip the chunk.
2. Return free space to OS, if viable.
OVERALL DESIGN

• Small chunks increase linked-list crawling overhead and fragmentation
• Large chunks possibly increase memory inefficiency

Usual solution: Set a threshold on small vs. big allocation.

• Small allocation:
  • Bucket the allocations according to $\log_2 [\text{size}]$
  • Use bitmaps of size smaller than the threshold
• Big allocation: Use separate chunk.
1. Find the category of the allocation.
2. If the allocation is small, try to find a free bitmap of the size and add the allocation.
3. If the allocation is big or a new bitmap is needed, allocate a new chunk
4. If there is no space left, ask OS for space and retry
5. If OS refuses to give more memory, fail.
1. Crawl through the list to find a chunk that contains the pointer for deallocation (chunks are intervals!)
2. Determine whether the chunk is a bitmap or single-block
3. Erase the block from the bitmap (if it’s a bitmap)
4. Erase the chunk if it is empty.
Assignment
ASSIGNMENT

Write an allocator that works on a static area of memory with known size.

• On initialization, the algorithm receives a continuous block of memory
• The algorithm sets up any required management structures on this memory
• For testing, the algorithm will be required to handle a set of allocate/deallocate requests from some simple algorithm.
• No OS communication will be required.
• Usual allocators are reentrant. Your solution is not required to be reentrant.
Use the standard C++ allocator interface.

```cpp
std::vector<int, some_allocator<int>> v;
```
/* declare a static description of the heap object */
struct heap_holder {
    static inblock_allocator_heap heap;
};

/* create the heap (this does not allocate memory!) */
inblock_allocator_heap heap_holder::heap;

/* assign some memory */
heap_holder::heap(0x6437856328, 10*1024*1024);

/* use in code */
std::vector<int, inblock_allocator<int, heap_holder>> v1;
std::vector<int, inblock_allocator<int, hh2>> v2,v3,v4;
class inblock_allocator_heap {
    // ...your static data here...
    void operator()(void* ptr, size_t n_bytes) { ... }
};

template<typename T, typename HeapHolder>
class inblock_allocator {
    // ...your solution here...
};

Wrap your solution in header file inblock_allocator.hpp.

If required, you can separate the solution into multiple header files and .cpp modules. Not required at all. Do not do it.
You can also pass dynamic allocator parameters using prepared structures in containers:

```cpp
class allocator_type;

explicit std::vector::vector
  (const allocator_type& alloc = allocator_type());
```

The static information will be copied among the allocators together with the allocator.

We will use the static approach.
• The algorithm **must only use the assigned memory area**
  • no `malloc`, `mmap`, `brk` or any other calls, from neither `inblock_allocator_heap` nor `inblock_allocator`
  • extra $O(1)$ of static storage allowed e.g. for storing the pointer to the assigned memory
CRITERIA

• The algorithm **must only use the assigned memory area**
  • **no malloc, mmap, brk** or any other calls, from neither
    inblock_allocator_heap nor inblock_allocator
  • extra $\mathcal{O}(1)$ of static storage allowed e.g. for storing the
    pointer to the assigned memory
• Allocation will correctly partition the assigned memory
  area among the requests
  • result of `allocate` will not overlap any other allocated
    memory
  • `deallocate` will reliably make the memory available for
    further use
  • all allocated addresses will be **aligned**
CRITERIA

• The algorithm **must only use the assigned memory area**
  • no **malloc, mmap, brk** or any other calls, from neither **inblock_allocator_heap** nor **inblock_allocator**
  • extra $\mathcal{O}(1)$ of static storage allowed e.g. for storing the pointer to the assigned memory
• Allocation will correctly partition the assigned memory area among the requests
  • result of **allocate** will not overlap any other allocated memory
  • **deallocate** will reliably make the memory available for further use
  • all allocated addresses will be **aligned**
• Support data structures should be reasonably efficient
  • Avoid fragmentation
  • Avoid large support structures
  • Test programs will use peak 33% of the ‘raw’ memory volume of the assigned memory area
• Do not try to beat the standard `malloc()`.
  (but try not to be 1000× slower)
• Available memory ‘heap’ will be relatively small (even with the 300% overhead!), be careful with the thresholds.
• Various optimizations that can help the performance&efficiency:
  • Bitmaps may carry a pointer to the next bitmap of the same block size
  • Heuristic to save memory: bitmap sizes may grow exponentially from a relatively small number
**HINTS — ALLOCATOR**

*Allocator named property* specifies the members of allocator-capable class that need to be present for the interoperation with rest of C++ library.

See

https://en.cppreference.com/w/cpp/named_req/Allocator

You are not required to implement obsolete or optional members, including:

- `A::template rebind<U>` — used for allocating different types
- `A::is_always_equal` — used for optimizations in some containers
- `A::propagate_on_container_{move,copy,swap}` — used for controlling the lifetime of allocator object
Various CPUs do various weird things if you access memory using unaligned pointers.

Pointer $p$ is aligned to $n$ iff

$$p \equiv 0 \pmod{n}$$

Align all memory addresses to avoid trouble. Recommended alignment is 8 bytes.
Optimality of your solution depends on a lot of heuristics.

If going with big vs. small chunks,

• you don’t know what is the optimal threshold to expect,
• any optimization on simple test cases can lead to problems with bigger cases.

Solution:

• Define the threshold as a constant so that we can change (and fix) it easily during testing.
• Aim for robustness, not optimality.
Many containers require implementation of additional allocator methods!

- `A::operator==(const A&)`
  decide whether allocator instances are compatible (used when e.g. moving containers)

- `template<typename U> A(const A<U>&)`
  copy-construct from a same kind of allocator for different type (used e.g. when containers need multiple data types)
Submit to ReCodex.

You should be able to see (and enroll to) the Advanced C++ group.

The task description, test programs, and copy of the slides will appear in ReCodex ASAP.
In this assignment, you have a relatively high chance of getting segmentation faults because of memory corruption.

Memory corruptions caused by allocators are nearly impossible to debug using standard means.
Time-saving advice: Write the program in small, simple steps; make sure that individual building blocks work correctly before progressing further.

For example:

1. The interface works, but cheats by only calling `malloc/free`.
2. The solution allocates consecutive blocks on the given memory heap, `deallocate` does not do anything.
3. The allocated blocks are formatted as chunks
4. The chunks may be found by a pointer and deallocated
5. Allocation can create bitmap chunks and select a viable bitmap chunk for adding new data, but bitmap chunks are never really removed
6. Bitmap chunks are correctly destroyed when the bitmap becomes empty.
Evaluation
• Program builds from source on major compilers
• Program does not crash, freeze, abort, hang, segfault, die, run into infinite loop, trigger OOM, throw an unhandled exception, cause undefined behavior, …
• Program does not leak any memory
• Test programs return the same results as with standard allocator
EVALUATION CRITERIA — CODE METRICS

- **less code is better**
- **easily readable code is better**
- consistent formatting (try astyle or clang-format)
- reasonable identifier names
- no magic constants
- comments
  - Hint: include a comprehensive “structure of solution” (SOS) comment on the top of the file
- C/C++-style efficiency measures
- `-Wall`, cppcheck (valgrind may not apply this time)
- portability to all major compilers
You may use bonus points to patch up some amount of point loss from minor/pedantic issues.

Optional bonuses:

- Optimized finding of the next chunk
- Optimized sizing of bitmaps
- Measurable improvements in bitmap implementation (avoid wasting instructions on individual bits)
- Performance better or comparable to \texttt{std::allocator}
- Structure better than chunks+bitmaps
- \texttt{[insert your brilliant idea here]}