Memory Profiling and Custom Allocators in C++

6CCS3PRJ Final Project Report

Author: Robert Bartlensky
Student ID: 1536771
Programme of Study: MSci Computer Science
Supervisor: Dr. Andrew Coles

April 2018

Originality avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Robert Bartlensky
April 2018

Abstract

Any non-trivial program needs to dynamically allocate memory. In C++, the `new` keyword is used to allocate memory on the heap. The compiler translates this keyword into a call to the standard function `operator new`, followed by a call to constructor of the allocated type. In C++, `operator new` is implemented using the standard C function `malloc`. The `malloc` function allocates additional space, called metadata, which is used to store a structure that allows the implementation to handle incoming allocation and deallocation requests quickly. Every allocation contributes to the size of the metadata. This overhead is negligible for large types, but for small objects it becomes a problem. If a program allocates a large number of small objects, then the overheads of `malloc` will occupy more space than the objects themselves.

The aim of this project is to create an efficient memory pool allocator, which minimizes the memory overheads of small allocations. This allocator will also be used to create a new implementation of `operator new` and `operator delete` which is going to store small allocations in memory pools by default.

Memory pools can only reduce memory usage when a sufficiently large number of small allocations is made. Therefore it is also necessary to create a profiling tool which will help decide if a memory pool might improve the memory usage of a program.

Acknowledgements

I would like to thank my supervisor, Dr. Andrew Coles, for providing advice and guidance throughout the project. His invaluable feedback enabled me to achieve all the goals of the project.

Contents

Chapter 1

Introduction

The aim of this report is to develop a framework for profiling and reducing the memory usage of C++ programs. Chapter 2 introduces memory pools, and other related concepts which help with understanding the implementation. Chapter 4 shows two possible memory pool designs which can be implemented as C++ allocators to achieve the goals of this project. Chapter 5 describes the implementation details of the framework, and the reason why the allocator, as described in chapter 4, is designed in such an unusual way. In order to demonstrate that the allocators perform as expected, chapter 7 and 8 show how the implementation compares to other popular memory pool allocators, and how it performs when included as part of a larger system, such as the Popf planner.

1.1 Motivation

Memory pools have always been used to reduce the memory usage of programs which allocate large numbers of small objects.

Cocos2d-x is a popular open-source library and game engine written in C++. It is used by many developers to build highly-efficient games that can be run on many platforms, including mobile phones. Games have high memory requirements, and mobile phones typically have less memory than other computer systems. The library has a built-in memory pooling system. Every object has a static `Create` method which places the object into a memory pool by default[1]. This feature enables developers to focus on building the game itself without having to worry about memory usage.

When building a game, the programmer must always use the `Create` method instead of `new`. This can be very confusing to beginners, who are used to allocating memory with `new`. The `Create` method is also responsible for managing the pointer it returns, which means developers do not need to free the pointers.

There are also multiple general-purpose libraries that implement memory pools such as *boost* [1] and *MemoryPool* [2]. Listing 1.1 shows how *MemoryPool*

[1] The method does other things as well, but such details are beyond the scope of this report.

can be used to allocate integers.

Listing 1.1: Allocating two integers using *MemoryPool*

```
MemoryPool<int> p;
int x1 = p.allocate();
int x2 = p.allocate();
```

In order to use *MemoryPool*, it is necessary to use a *MemoryPool* instance to allocate and deallocate memory.

The main goal of this project is to come up with a general-purpouse pool allocator similar to *boost* and *MemoryPool*, that performs better than `malloc` when allocting small objects. Another important goal is to reimplement `operator new` and `operator delete`. The new allocator should minimize the memory overheads of all allocations. In order to make the new implementation available to programs, an injection mechanism must be developed as well. Its purpose is to automatically insert the reimplemented operators into programs, such that when they use `new`, the objects are placed into memory pools by default, similarly to what Cocos2d-x does. Chapter 8 describes in detail why in some cases the injection mechanism reduces the memory usage of programs, while in other cases it increases the memory usage substantially.

1.2 Why C++?

C++ is a unique programming language, as it combines low-level features which enable direct memory manipulation with high-level constructs, such as classes or lambdas. The low-level features of C++ are very important in the context of this project, because the allocators rely on manipulating memory to achieve reduced memory usage, and quick allocation and deallocation times.

The language also allows operators, such as `operator new` to be overriden, which makes the task of injecting memory pools into programs easier.

C++ is a very important and widely used programming language. A large number of applications are implemented in C++, because of its high execution speed and object-oriented features. Most large-scale applications could benefit from using memory pools, but it is often infeasible to change an older code base to allow objects to be created in pools. An automatic injection mechanism would provide an easy way to integrate memory pools into existing programs.

Chapter 2

Background

This chapter describes a few important concepts which are used to design and implement the allocators described in this report. It is important to understand the memory layout of a process, and which parts of it can be improved. A simplified `malloc` algorithm is presented in order to illustrate why memory pools are important and useful. The concept of memory pools is also introduced, but these are discussed in more detail in chapter 4. In chapter 5 an injection mechanism implemented in LLVM is presented, thus it is necessary to describe what LLVM is, and how it can be used in the context of this project.

2.1 The memory of a process

Before introducing the concept of memory pools, it is important to understand the distinction between the different types of memory that a process uses.

The memory of a process is split into 5 parts: *text*, *data*, *bss*, *heap* and *stack*. The *text* section is where the machine instructions of the program are stored. All the data stored in this section is read-only. The *data* and *bss* sections store global and static variables. The main difference between them is that the *data* segment stores variables that have been initialized, and the *bss* stores uninitialized variables.

2.1.1 The stack

This is the region of a process' memory which is used to create stack frames. The *stack pointer* is a register which holds the address of the top of the stack. Whenever a function is called, a new stack frame is created by decreasing the value of this pointer. This creates space for the function's parameters and local variables. When the function returns, the stack frame is destroyed by increasing the value of the stack pointer. The programmer has no control over this region[1].

[1]The `alloca` function can be used to allocate stack memory, but as soon as the function which called `alloca` returns, the allocated region is deallocated. More information can be

2.1.2 The heap

This is the memory area where dynamically allocated objects are stored. In C and C++, the heap is managed by `malloc`, `free`, and `realloc`.

2.1.3 Improving memory usage

The memory layout of most segments cannot be improved at run-time. For instance, the *text*, *data*, *bss*, and *stack* should not be manipulated, as there is nothing users can do to improve their layout. The sizes of these sections are determined at compile time. Thus, the compiler is responsible for optimizing the memory usage of these regions.

The only segment left is the *heap*, which is managed by `malloc` and `free`. Because both of these functions are loaded during the execution of a program, that means it is possible to come up with a better heap allocation strategy, and link it to the program in order to reduce the memory usage.

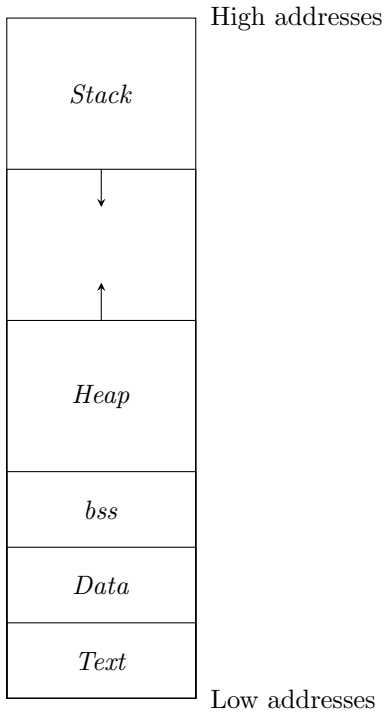

Figure 2.1: The memory of a process

Figure 2.1 shows the memory layout of a process. All segements have a fixed size except the for the stack and the heap, whose lengths change during the

found at `http://man7.org/linux/man-pages/man3/alloca.3.html`.

execution of the program.

2.2 Allocators

Allocators are used to allocate, deallocate, construct, and destruct objects of a certain type [3]. In C++, allocators are used to implement containers such as *std::vector*. Their responsibility is to allocate a number of contiguous objects of a certain type, and to construct them.

This project involves creating an efficient memory pool allocator. Unfortunately, due to the design of the implementation, the memory pools described in this report are not compatible with the definition of a C++ allocator. This is due to the fact that the implementation cannot allocate any number of contiguous objects, which a C++ allocator must be able to do.

2.3 Heap allocation

In C++, the `new` keyword is used to allocate objects on the heap. The compiler translates the code shown in listing 2.1 into the instructions shown in listing 2.2.

Listing 2.1: A simple heap allocation program.

```
new SomeObject(arg1, arg2);
```

Listing 2.2: The translation of listing 2.1 by a C++ compiler.

```
auto p = static_cast<SomeObject*>(::operator new(sizeof(SomeObject)));
p->SomeObject(arg1, arg2);
```

The actual memory allocation is performed by `operator new`. In the GNU implementation of C++, `operator new` calls `malloc` in order to allocate memory on the heap.

In order to improve memory usage, it is important to first understand why the default `malloc` implementation does not allocate memory optimally. The first question to answer before attempting to create an efficient allocator is: how does malloc work?

2.3.1 Memory Control Blocks

Memory Control Blocks (MCBs) are the building blocks of `malloc`. An MCB is a structure which is used to partition the heap into memory blocks. A block of memory is shown in figure 2.2.

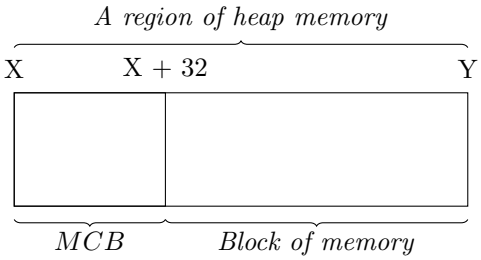

Figure 2.2: An MCB followed by a block of memory.

MCBs are the nodes of a doubly linked list. The heap is divided into memory regions. Each region is composed of an MCB, followed by a block of usable memory. Two blocks are linked through their respective MCBs. MCBs also keep track of the size of the block, and a flag which denotes if the block is occupied or not. Listing 2.3 shows a C++ `struct` that implements an MCB.

Listing 2.3: The MCB structure.

```
struct MCB {
  MCB *next, *previous; // the doubly linked list structure
  size_t size; // the size of this block measured in bytes
  bool isFree; // if the block is used or not
};
```

The MCB structure has a size of 32 bytes and is 8 bytes aligned. The structure should technically occupy 25 bytes, but because the alignment is 8, the size is padded with zeroes until the next multiple of the alignment is reached, which is 32 in this case. The alignment of a structure is exaplined in detail in section 2.5. The size of this structure is used in later sections to show the overheads of `malloc`.

2.3.2 A simplified malloc algorithm

The initialization phase of the `malloc` algorithm uses system calls in order to allocate heap memory for the process. The algorithm allocates 4 KiB of memory (i.e. one page)[2]. Next, the algorithm creates an MCB at the start of the memory block returned by the Operating System (OS).

The `next` and `previous` fields of the first MCB are set to `nullptr`, the `size` field is set to $4\ KiB - sizeof(MCB) = 4064$, and `isFree` is set to `true`.

This phase is usually executed before the `main` function runs. Figure 2.3 show a representation of the state of `malloc` after the initialization phase.

[2]The actual `malloc` algorithm allocates more memory, but for simplicity, it is assumed that the algorithm always allocates a page of memory.

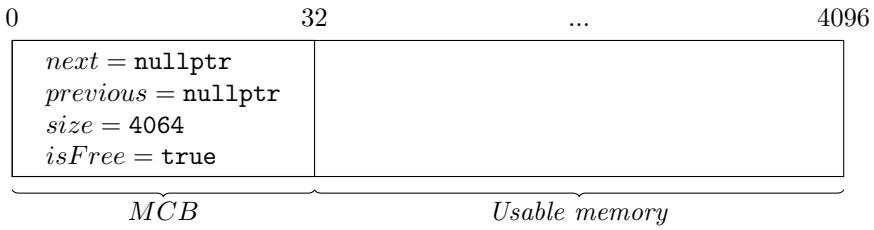

Figure 2.3: The state of `malloc` after the initialization phase.

To understand what happens next, consider the program shown in listing 2.4 which allocates an integer on the heap by using `malloc`.

Listing 2.4: An integer allocated with `malloc`.

```
int* x = static_cast<int*>(malloc(sizeof(int)));
```

The size of an integer is 4 bytes, which is why `malloc` will try to create a block of size 4 and return it to the user. In order to satisfy the request, the block that was created during the initialization phase is split into two distinct blocks.

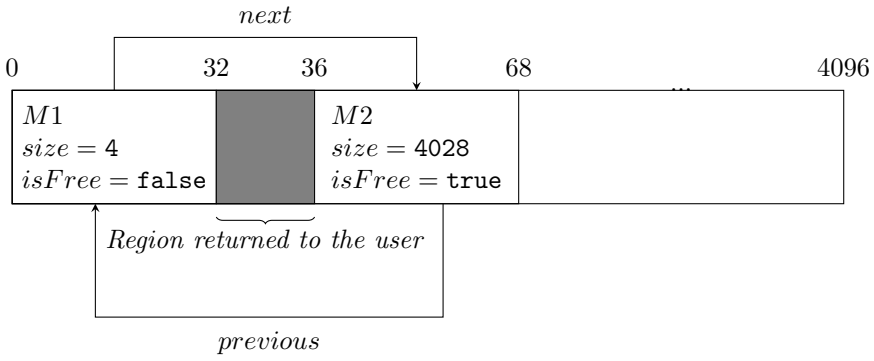

Figure 2.4: The split memory block.

Figure 2.4 illustrates how `malloc` splits the initial block. A new MCB (*M2*) is created at the address located at 4 bytes after the first MCB (*M1*). The size of *M1* is set to 4, and it is no longer free. The size of *M2* is set to 4028, because that is how much usable memory is left. The `next` field of *M1* is set to point to *MC2*, and the `previous` field of *MC2* will point to *MC1*. The remaining fields remain set to their default values. After the block is split, `malloc` returns a pointer to the new memory block of size 4, shown as a gray rectangle in figure 2.4. In the case where the last available slot is not large enough to satisfy the allocation request, the algorithm allocates a new block of memory, and links it to the other blocks. This is done by using the initialization algorithm again

to allocate a page of memory, but this new block is then linked to the other blocks.

Deallocation works in a similar way, but blocks are merged instead of being split. When `free` is called on a pointer, the MCB that is 32 bytes to the left in memory sets its `isFree` member to `true`. If there is an adjacent MCB that is also free, then the two MCBs are merged. If the integer from figure 2.4 is deallocated, then the two MCBs are merged, and the memory block will look like it did previously in figure 2.3.

2.3.3 Overheads of malloc

The `malloc` function wastes a lot of memory for small objects. The implementation starts with 4064 bytes of usable memory, and, after an integer allocation, this number drops to 4028 bytes. For every integer allocation, an additional 32 bytes are used solely for storing MCBs. Ideally, these wasted bytes could be used instead to allocate more integers. Although this is a problem when allocating small objects, these overheads become negligible for larger allocations.

2.4 Memory pools

Memory pools provide a way to minimize the allocation overheads by grouping allocations in a block of memory, also called a memory pool.

2.4.1 A pool of objects

A pool is a contiguous block of memory which is partitioned in such a way that it can fit N objects of a certain type with little to no overheads. Pools can be viewed as arrays. Figure 2.5 illustrates a memory pool for integers.

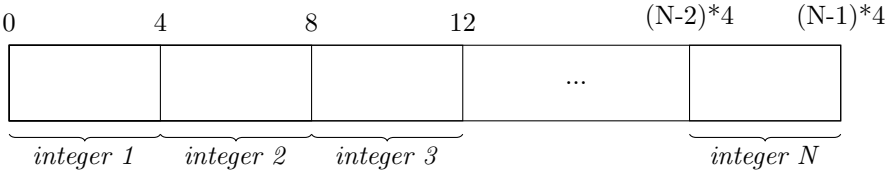

Figure 2.5: A pool of N integers.

There are two issues that need to be addressed when designing a memory pool. The first one is that there is no way to tell if a slot is free without using an additional structure to store this information. The second issue is related to what happens when there are no free slots left. There is also another problem which is related to the alignment of the slots, but this is discussed in a later section.

2.5 Alignment

Each data type has an alignment. If a type has an alignment of N, it means that any address that points to an object of this type must be divisible by N. If objects are misaligned, then programs will run much slower [4]. One property of the alignment of a pointer is that is it always a power of two. Another important property is that if a pointer is 16 bytes aligned, then it is also 2, 4, and 8 bytes aligned as well[3]. One of the important optimizations described in chapter 5 is based on the first property. It also worth mentioning that the maximum alignment in C and C++ is 16 bytes[4].

As an example, consider the structure shown in listing 2.5:

Listing 2.5: A *Foo* structure.

```
struct Foo {
  int x; // alignment of 4
  bool y; // alignment of 1
}; // alignment of 4 because we take the maximum of the alignments
```

The *Foo* struct has to be 4 bytes aligned, which means that any pointer to a *Foo* object must be divisible by 4. For example, the address *0x00000004* is a correctly aligned pointer to an object of type *Foo*, while *0x00000005* is not because $5 \; mod \; 4 = 1$. Consider a pool of *Foo* objects.

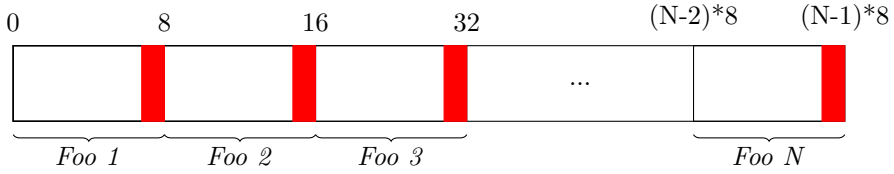

Figure 2.6: Memory pool for *Foo* objects.

The red regions represent the padding of each *Foo* object. The padding is necessary in order to properly align objects. It can also be viewed as lost or unusable memory, but there is no efficient way around it. *Foo* objects require at least 5 bytes of memory to store all their members[5]. Because *Foo* has an alignment of 4, all *Foo* pointers must point to an address whose value is a multiple of 4. The smallest multiple of 4 that is greater than 5 is 8. This means that 3 bytes are lost due to padding.

Both memory pools, and `malloc` will waste some memory in order to align objects properly.

[3] Because 2, 4 and 8 are divisors of 16.

[4] Users can set the alignment of a structure to be any power of two by using the `alignas` keyword.

[5] `sizeof(int) + sizeof(bool) = 5`

2.6 LLVM

LLVM[6] is an infrastructure that can be used to implement programming languages. In order to implement a programming language, the user has to translate their language into LLVM bytecode. The LLVM infrastructure is responsible for optimizing, and generating machine code out of the produced bytecode.

The LLVM bytecode is composed of a module. Each module has a number of functions, and global variables. Functions are composed of multiple basic blocks which are sets of instructions. The LLVM infrastructure can also be used as a library to parse, and alter LLVM bytecode.

LLVM optimizes the bytecode by running passes. Each pass is responsible for performing an optimization, for instance, a pass might be used to carry out dead code elimination. There are many types of passes, for example, basic block passes, which enable the user to modify the contents of basic blocks. It is important to know that a pass is run on every basic block, and the code must not modify any other basic blocks except for the one that is being processed. In chapter 5, a basic block pass is used to modify *call* instructions such that calls to `new` are replaced with calls to the allocator that is described in this report. LLVM allows user defined passes to be run together with its own set of optimization passes.

2.7 Relevant literature

The most important piece of literature is the `malloc` implementation by Doug Lea [5]. This implementation differs from what was described in section 2.3.2 because it is more complicated. Conceptually it uses MCBs, but the `next` and `previous` fields point to MCBs of the same size rather than to neighbouring ones. This is because the algorithm allocates multiple linked lists for different allocation sizes. For instance there are bins for objects that have a size that is lower than 8, or lower than 256. This approach minimizes the overheads of smaller objects from 32 bytes to somewhere around 8 bytes. The larger the allocation is, the larger the overheads are [6].

There are several papers [7] and websites [8] which describe other memory pool implementations, but the problem with them is that the user has little to no control over the way objects are placed into pools. One paper describes how memory pooling can be done at compile-time, but this project is focused on a run-time pool allocator [7]. Other websites show how users can implement memory pools for certain objects, but they do not describe a general approach [8]. There are also a large number of memory pool implementations on GitHub.

In order to implement these memory pools as efficiently as possible, it is important to understand how operators `new` and `delete` work, and also how allocators are implemented. The GNU implementation of *libstdc++* was one of the most useful resources when developing the implementation described in this report. It provided the source code of `new` and `delete`, which proved to

[6] `https://llvm.org/`

be very useful, since one of the tasks of the project is to reimplement these two operators.

Websites such as CppReference[7] also provided useful information regarding all the different types of `new` [9] and `delete` [10] operations, which needed to be overriden in order to implement a fully-functional global memory pool allocator.

[7] `http://en.cppreference.com/`

Chapter 3

Requirements

There are several requirements that the allocators, injection tools, and profiling tools must meet. Some of these are related to their performance, while others are concerned with their usability. How these requirements are met is described in chapters 4, 5, and 7.

3.1 Performance requirements

In terms of performance, the allocators must meet the following conditions:

1. The allocation, and deallocation times need to be faster than `malloc` in order to ensure that the allocators do not become the bottlenecks of the programs that use them.
2. The metadata of allocations should be minimized. The `malloc` algorithm has an 8 bytes overhead per small object. This means that the allocators described in this report must not use more than 7 bytes of metadata per object, because otherwise memory usage is not improved.

3.2 User requirements

The allocators should:

1. Be easy to use by providing a simple interface. This enables users to easily integrate memory pools into their systems.
2. Come in the form of a shared or static library which can be linked with a program to allow users to use the allocators.

The injection mechanism should:

1. Provide a way to allocate objects in pools without needing to modify the system's source code.

2. Aim to reduce the memory usage of the program.

There are programs that might not allocate large numbers of small objects, in which case, the pool allocators are not going to help reduce the memory usage. A memory profiling tool must also be implemented in order to gather this information. The tool should:

1. Collect allocation information, such as the number of times all types are allocated, which functions allocate objects, the peak number of objects, etc.
2. Store the information in a sensible format when the execution of the program ends.
3. Provide a user iterface that displays the information in a table. Users of the tool should be able to:
   (a) Search for a certain type or function, and filter out the rest.
   (b) Sort the data in ascending or descending order by type name, number of objects allocated, function names, etc.

Chapter 4

Design

This chapter describes two memory pool designs called *BitPool* and *LinkedPool*, and the design of a global allocator which can be used as a replacement for `new`, and `delete`.

4.1 Pool header

Both designs work very similarly, but there are some key differences between them which are highlighted in later sections. In section 2.4.1, figure 2.5 shows how a pool of integers might be stored in memory. Chapter 2 mentioned that any memory pool implementation must be able to determine if a slot is empty or not. The two designs solve this issue by allocating extra space at the beginning of the memory block, and creating a structure called a *pool header*.

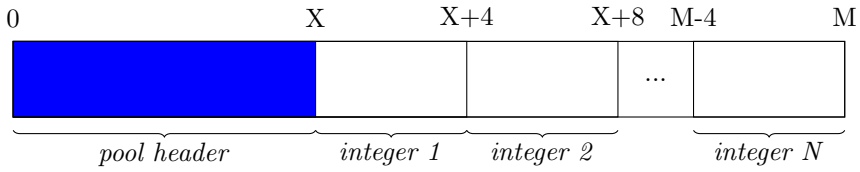

Figure 4.1: A pool header followed by N integers.

The pool header is responsible for keeping track of the availability of the slots. The two designs use the pool header differently, as described in the sections that follow. Note that the header in figure 4.1 depicts a header of a generic pool implementation. The size of a header does not have predefined size, because it depends on the implementation.

4.2 Memory allocation

Before discussing how object pools can use the pool header to keep track of free slots, it is necessary to describe how pools are created. Consider the following UML diagram:

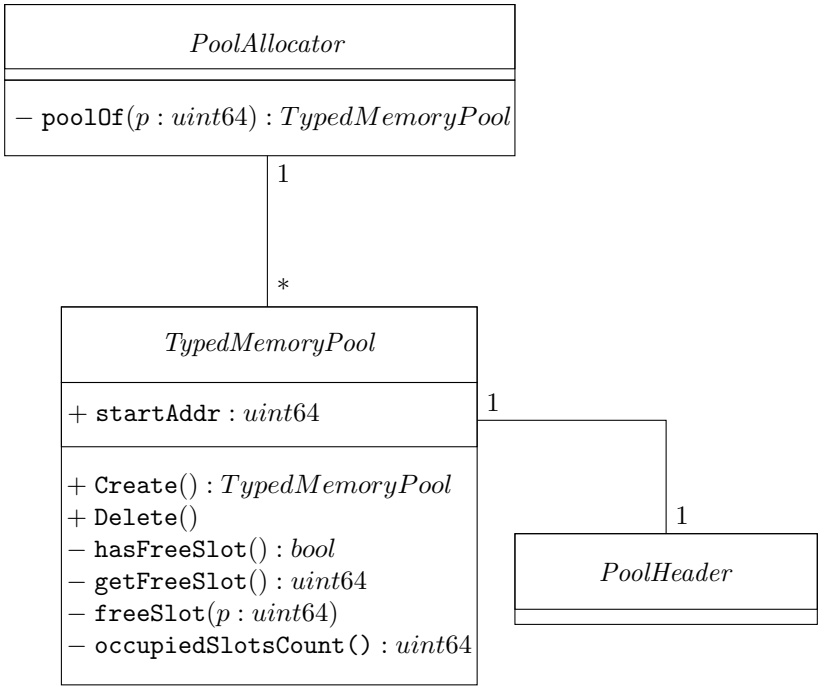

Figure 4.2: UML diagram showing the general structure of a *PoolAllocator*.

Figure 4.2 shows the general structure of a memory pool allocator that uses pool headers to track the state of slots. A *PoolAllocator* creates *TypedMemoryPool*s in which it stores a *PoolHeader*, and the slots of the pool. Both *BitPool*, and *LinkedPool* implement the `hasFreeSlot`, `getFreeSlot`, `freeSlot`, and `occupiedSlotCount` methods from figure 4.2 differently. Their implementation is discussed in later sections. The other methods can be implemented independently of any design choices. Each allocator must also define the data that it stores in *PoolHeader*.

The `Create` method creates a block of memory in which the allocator can store the header, and the slots. This method allocates a page of memory[1], and stores the start address of the block in the `startAddr` field. This is used to allocate a pool of free slots when an allocation request is processed, and there are no available slots. The `Delete` method deallocates the block of memory which starts at the address specified by `startAddr`. These two methods can

[1]The size of a page is usually 4 KiBs = 4096 bytes.

be implemented with `malloc`, and `free`. The `malloc` function can be used to allocate a page of memory, which is later deallocated with `free`. When pools become empty, they are deallocated in order to save memory.

The `poolOf` method takes a pointer as an argument, and returns the pool from which the pointer originates. Listing 4.1 shows a general `poolOf` algorithm.

Listing 4.1: The `poolOf` algorithm.

```
Algorithm poolOf(p : uint64) : TypedMemoryPool
  Let MP = the list of the allocated TypedMemoryPools
  for each pool ∈ MP:
    if p ≥ pool.startAddr ∧ p ≤ pool.startAddr + 4096:
      return pool
  return ∅
```

The `poolOf` algorithm loops through all the pools, and checks whether the argument is in the range of addresses of one of the pools. The check is done by comparing the argument with the start address, and the end address of each pool. The size of a *TypedMemoryPools* is always 4 KiBs, thus the end address is calculated by adding this number to the start address. The reason why the size of a *TypedMemoryPool* is 4 KiBs is explained in chapter 5.

The `allocate`, and `deallocate` methods are implemented by using all the methods provided by the *PoolAllocator* and *TypedMemoryPool* classes. Listings 4.2, and 4.3 describe general memory pool algorithms for allocation and deallocation.

Listing 4.2: A general pool allocation algorithm.

```
Algorithm allocate : uint64
  Let MP = list of the allocated TypedMemoryPools
  for each pool ∈ MP:
    if pool.hasFreeSlot():
      return pool.getFreeSlot()
    else:
      P = TypedMemoryPool.Create()
      MP = MP ∪ P
      return P.getFreeSlot()
```

Listing 4.3: A general pool deallocation algorithm.

```
Algorithm deallocate(ptr : uint64)
  Let MP = list of the allocated TypedMemoryPools;
  P = poolOf(ptr)
  P.freeSlot(ptr)
  if P.occupiedSlotsCount() = 0
    MP = MP \ P
    P.Delete()
```

4.3 BitPool

BitPool is a *PoolAllocator* that uses bits to represent the state of each slot of a *TypedMemoryPool*. These bits are stored in the *PoolHeader*, along with a counter which denotes the number of free slots in the pool. The first 8 bits of the pool header encode the binary representation of the counter, and the next N bits represent the state of the N slots of the memory pool.

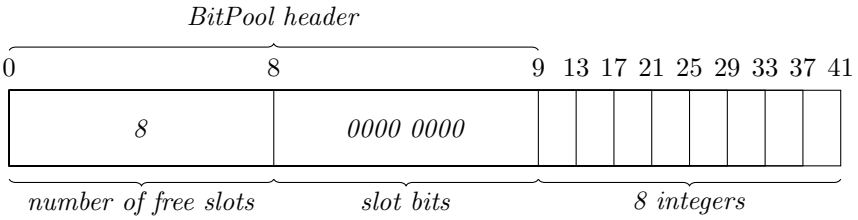

Figure 4.3: A bit pool of 8 free integer slots.

As described above, each bit represents the state of a slot. In figure 4.3, the first high-order bit (i.e. counting from left to right) denotes the state of the first slot, which is free in this case. The counter is set to 8 because there are 8 free integer slots. Whenever a slot is allocated, the counter is decremented, and the bit which represents the allocated slot is set to 1. Consider the case where slots 1, 3 and 8 are occupied.

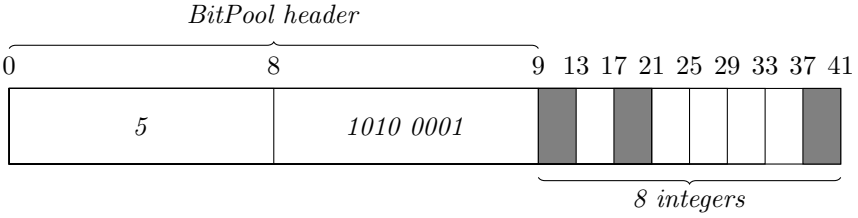

Figure 4.4: A bit pool where slots 2, 4, 5, 6, 7 are free.

In this example, the number of free slots is now 5, because only 5 of the slots are free. The bits which represent the slots 1, 3, and 8 have all been set to 1, while the rest of the bits remain equal to 0. If the third slot was freed, then the slot bits would change from 1010 0001 to 1000 0001.

4.3.1 Pool header size

The counter can be stored as a *size_t*, which has a size of 8 bytes. The state of the slots is stored as a sequence of bits. In the example given in figure 4.3, there are 8 integer slots, therefore 1 byte of memory is used to track the state of the 8 slots. In the case where there are 9 slots, the header must allocate space for

2 bytes because memory is byte-addressable. As a result, the size of the header depends on the type that is stored in the pool.

4.3.2 Implementing the PoolAllocator interface

hasFreeSlot

This method checks whether the counter of a *PoolHeader* is greater than 0, and returns the result.

getFreeSlot

BitPool looks for a bit that is set to 0, changes the value to 1, and returns the address of the slot which corresponds to the bit. It is possible to determine the address of the slot that needs to be returned by using the following formula:

$$address = PoolHeaderSize + sizeof(Type) * bitPos \tag{4.1}$$

where $Type$ is the type of data stored in the memory pool, and *bitPos* is the position of the bit in the sequence of slot bits. Once the address is calculated, it can be returned to the user.

freeSlot

This method sets the availability of a slot to free. The implementation finds the bit which corresponds to the slot that is being freed, and sets it to 0. This can be done by extracting *bitPos* from equation 4.1. Users can no longer use slots after they are freed.

$$positionOfTheBit = \frac{addrOfSlot - PoolHeaderSize}{sizeof(Type)}$$

occupiedSlotsCount

The header can be used to implement this method. The *counter* field denotes the number of free slots in a pool. The number of occupied slots can be calculated by subtracting the counter from the total number of slots.

4.4 LinkedPool

The main difference between *LinkedPool* and *BitPool* is the way the two implementations keep track of free slots. Unlike *BitPool*, *LinkedPool* keeps track of free slots using a linked list. Many pool implementations [1, 2] use this approach, but this design does it in an unusual way. Listing 4.4 below describes how a linked list node is structured.

Listing 4.4: A linked list node.

```
struct Node {
  Node* next;
}; // size 8, alignment 8
```

A *Node* has a pointer to the next node, and does not keep track of any other information. Each *TypedMemoryPool* has its own linked list, similarly to *BitPool* where every pool kepps track of slot bits. The *PoolHeader* in this case keeps track of the number of occupied slots, and the head of the linked list.

4.4.1 The hidden linked list

The linked list structure is supposed to keep track of all slots that are free. A node is creatd for each free slot, and is inserted into the list. Searching for a free slot involves returning the `next` member of the head of the linked list, which is stored in the header. Although a *Node* does not keep track of anything else other than the next element of the list, *LinkedPool* must be able to retrieve the address of the slot represented by the node. During the initialization of the *PoolHeader*, the allocator creates a *Node* object at the start address of each slot, and links it to the list. This is the reason why this linked list is said to be *hidden*. Consider the *LinkedPool* for objects of size 16 show in figure 4.5. The gray rectangles denote a linked list node, and the arrows show how the nodes are linked.

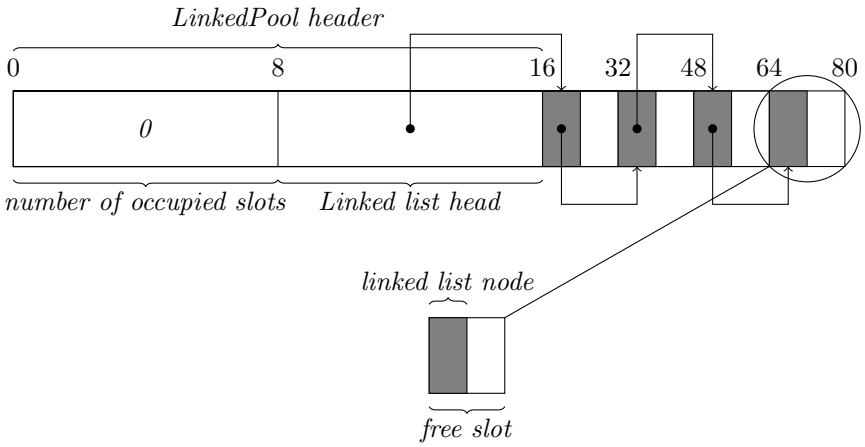

Figure 4.5: A linked pool which contains four free slots.

As a further explanation, the first slot starts at byte 16. Thus, during the initialization phase, *LinkedPool* creates a linked list node at that address, and links it to the *Node* starting at byte 32. Because nodes are created in this way, `head.next` encodes two things: whether there is a free slot in the list, and the address of the slot.

Consider the case where slots 1 and 4 are occupied.

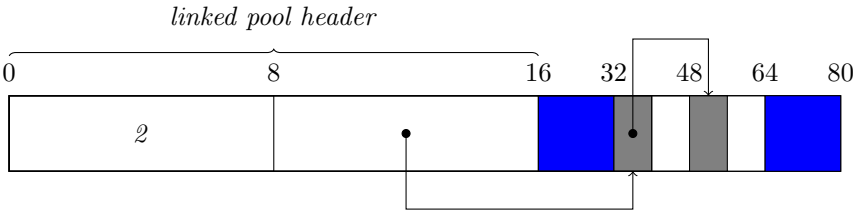

Figure 4.6: A linked pool where slots 2 and 3 are free.

In figure 4.6, the blue regions are occupied slots. Note that when all slots are occupied, the linked list is empty[2]. Also, the head of the list, which is part of the pool header, always points to a free slot, or a `nullptr`, if there are none. The number of occupied slots, which in the example from figure 4.6 is 2, is also recorded in the *PoolHeader*. When all slots are full, the head points to `nullptr`. When a slot is freed, a *Node* object is created at the beginning of the freed slot. This new *Node* is inserted between `head`, and `head.next`. Consider the case where the fourth slot is freed.

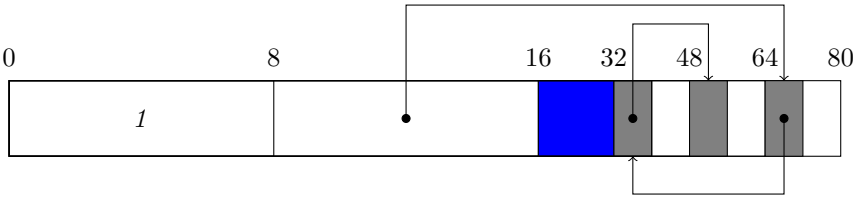

Figure 4.7: A linked pool after the fourth slot is freed.

Figure 4.7 shows that the linked list node that was created in the fourth slot was inserted after the head, between head and `head.next`.

4.4.2 Pool header size

The size of the *PoolHeader* is always 16 bytes, as shown in listing 4.5.

Listing 4.5: The *LinkedPoolHeader* struct.

```
struct LinkedPoolHeader {
  size_t occupiedSlots; // 8 bytes
  Node head; // 8 bytes
}; // 16 bytes in total
```

In this case, the *PoolHeader* has a constant size, which is independent of the type of objects stored in the pool.

[2]Except for the head of the list which is always part of the pool header.

4.4.3 Implementing the PoolAllocator interface

hasFreeSlot

This method checks whether `head.next` is a `nullptr` or not, and returns the result.

getFreeSlot

The address of a free slot is always stored in the `next` field of the `head` of the linked list. Thus, this can be implemented by returning `head.next`.

freeSlot

The implementation of this function is best described with an algorithm which is shown in listing 4.6.

Listing 4.6: The algorithm of the `freeSlot` function

```
Algorithm freeSlot(ptr: uint64)
  N = create a new Node at the address ptr
  if head.next exists:
    N.next = head.next->next;
  head.next = N;
```

A new node is create at the address of the slot that is freed, then it is inserted after the `head` of the linked list.

occupiedSlotsCount

This method is implemented by returning the counter of the header.

4.5 Comparing BitPool and LinkedPool

Both designs can be used to implement a memory pool allocator. This section compares the two approaches in terms of memory overheads and ease of implementation.

4.5.1 Overheads and alignment

Both implementations have to keep track of free slots through a *PoolHeader*. This header contains a counter which denotes the number of free slots in the case of *BitPool*, or occupied slots in the case of *LinkedPool*. In the case of *BitPool*, the header contains a bit for each slot, while *LinkedPool* stores the head of a linked list. Section 4.2 specified that `malloc` would be used to allocate pages of memory. Both implementations allocate a page of memory for a *TypedMemoryPool*, in which they store the header, and the slots. Therefore, the number of slots is determined by the size of the header. If the header is large, then

there si less space to store pool slots. In order to decide which pool is easier to implement, but also more efficient, it is only necessary to compare the two types of headers.

The size of a *LinkedPool* header is always 16 bytes. The two headers are equal if the slot bits occupy 8 bytes. This means that the *BitPool* header can track the state of at most 64 slots. If a pool is a page in size, this means that there are 4080 usable bytes which can be used to allocate slots. The remaining memory can be partitioned into 64 slots of size 63, as seen in the equation below.

$$\left\lfloor \frac{4096 - 16}{64} \right\rfloor = 63$$

Thus, for objects of size 63, both implementations have the same overheads per object, because they have headers of the same size. The following equations show how the overheads can be calculated.

$$\begin{aligned}
&P = 4096;\ \textit{the size of a } TypedMemoryPool \\
&H = 16;\ \textit{header size} \\
&R = P - H = 4080;\ \textit{usable memory} \\
&S = 63;\ \textit{the size of a slot} \\
&N = \left\lfloor \frac{R}{S} \right\rfloor = 64\ \textit{slots}; \\
&U = R - N * S = 4080 - 4032 = 48;\ \textit{unusable memory} \\
&O = \frac{U + H}{N} = \frac{64}{64} = 1\ \textit{byte};\ \textit{overhead per object}
\end{aligned}$$

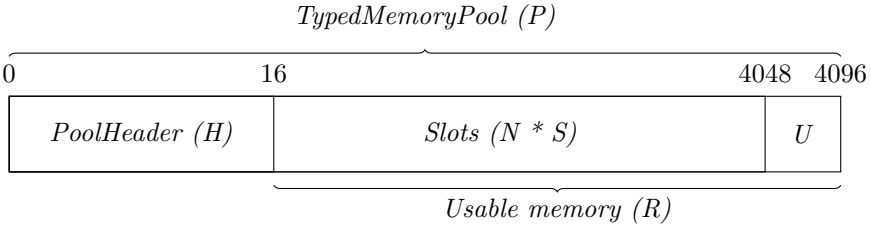

Figure 4.8: A pool with slots of size 63 bytes.

The overhead per object is 1 byte which is a lot smaller than the overhead of `malloc` which is 8 bytes in this case.

Before analyzing the overheads of memory pools for objects of size greater and smaller than 63, there are a few issues that need to be addressed. *LinkedPool* has one obvious shortcoming. For objects whose size is less than 8 bytes, the

implementation must create slots of 8 bytes in size. This is because the size of a *Node* is 8 bytes. In order to initialize a linked list node in all free slots, each slot must be at least 8 bytes in size. This means that memory is wasted for pools of a type whose size is smaller than 8 bytes.

Another issue to consider is the alignment of the pointers returned by the *PoolAllocator*. Slots must be aligned and padded properly based on the type stored in the pool. The first slot of a *LinkedPool* is always 16 bytes aligned, because the header has a size of 16 bytes, and, therefore, the first slot starts at an address divisible by 16. *BitPool*'s header on the other hand must be aligned and padded such that the first slot is correctly aligned. The other slots are padded if and only if the size of the slot is not a multiple of the alignment. If the size of an object is 24, but its alignment is 16, then slots must be padded until they reach a size of 32 bytes. In this case, the first slot starts at byte 16, the second slot at 32 + 16 bytes, and so on.

Consider a type whose size is 88 bytes, which is greater than 63. A *LinkedPool* can store $\left\lfloor \frac{(4096-16)}{88} \right\rfloor = 46$ objects of this type. Because $4080 \; mod \; 88 = 32$, this means that the pool wastes another 32 bytes in addition to the 16 bytes required to store the header. This is because the remaining memory cannot be split perfectly into slots of size 88. The total amount of wasted memory is therefore 48 bytes, 16 bytes for the header, and another 32 bytes that are left over from partitioning the memory into slots. The total amount of overhead per object is:

$$O = \frac{48}{S} = 1.04 \; bytes;$$
$$S = \left\lfloor \frac{4080}{88} \right\rfloor ; the \; number \; of \; slots$$

This is also less than the overhead of `malloc`. Because the size is a multiple of 8, this means that the slots are 8 bytes aligned. If this type needs to be 16 bytes aligned, then the slot size will be padded until it reaches a size of 96 bytes, which is a multiple of 16. In this example, if the alignment is 16, a lot of memory is wasted. Each slot is padded with 8 bytes, which means that the overhead of an object is at least 8 bytes.

For the type described above, *BitPool* can store 46 slots as well. The header in this case is 8 bytes + 46 bits to represent the 46 slots. The 46 bits can be represented by using 6 bytes, which is why the total size is 14 bytes. If the type has an alignment greater than 2, the header must be padded. If the type is 16 bytes aligned, then *BitPool* will also waste at least 8 bytes per object, due to padding.

Now consider a type whose size is less than 63 bytes. For example, a type whose size is 34 bytes may need to be stored in one of the two memory pools described in this report. *LinkedPool* allocates 120 slots, and the overheads per object are 0.26 bytes[3]. On the other hand, *BitPool* can only allocate 119 slots,

[3]Assuming an alignment of 2, which means slot sizes are 34 bytes in size.

with a header size of 23 bytes. The header must be padded in this case if the alignment of the type is greater than 1. If the alignment is 2, 4 or 8, the header must be padded until the slot size reaches 24 bytes. The pool wastes another 26 bytes due to the size of the type not dividing the usable memory perfectly, therefore it has an overhead of 0.42 bytes per object. If the alignment of the type is 16, then in both cases the slot size becomes 48 bytes due to padding.

Consider the case where a pool of integers is created. *BitPool* can create 990 slots, and each integer will have an overhead of 0.13 bytes. Because of the issue mentioned above, *LinkedPool* can only allocate 510 slots, with an overhead of 4.03 bytes per integer. In this case, the linked pool allocator has a higher overhead per object than *BitPool*, however its overheads are lower than the 8 byte overhead of `malloc`.

4.5.2 Time complexity

For objects of any size, the two implementations have similar overheads, except for sizes of less than 8 bytes, where *BitPool* has smaller overheads per object than *LinkedPool*. However, it is also important to compare the two allocators based on the complexity of the operations defined in section 4.2.

Both allocators implement the following the functions `hasFreeSlot`, `freeSlot`, and `occupiedSlotsCount` in constant time.

The two designs share the implementation of the `poolOf` function, which runs in $O(n)$. The main difference between the two allocators is the implementation of the `getFreeSlot` operation.

The linked pool allocator implements `getFreeSlot` in constant time by simply returning `head.next`. *BitPool*, on the other, hand implements this operation in *O(S)* time, where S is the number of slots of the memory pool. This is because the implementation has to look for a bit that is set to 0. Thus, in the worst case, it has to check all bit slots.

4.5.3 Conclusion

Both allocators can be used to implement memory pools. The implementation of *BitPool* is not described in chapter 5. This is due to the fact that the design of *LinkedPool* can be translated more easily to a high-level language such as C++. This is because the slot bits are harder to implement than the hidden linked list. Another reason why *LinkedPool* is implemented instead of *BitPool* is because the functions described in section 4.2 have a better worst case running time in the case of *LinkedPool*. Even though *LinkedPool* wastes more space than *BitPool* on types whose size is smaller than 8 bytes, allocations of such sizes are rare in large systems.

4.6 A custom operator new and delete

This section describes the design of an allocator that can replace the implementation of `operator new` and `operator delete`. The design of *LinkedPool* will play an important role in implementing a replacement for these two operators.

One thing to note is that the pool allocators described in section 4.2 cannot be used for types that occupy a lot of memory. This is due to the fact that a *TypedMemoryPool* is always limited to a page in size. Therefore, objects that are large will not fit, or will cause lots of memory to be wasted. Consider a type whose size is 3000 bytes. *LinkedPool* can only allocate a single slot for this type, and, therefore, there is a loss of 1086 bytes, which is very memory inneficient. The `malloc` function can satisfy this allocation by only wasting 32 bytes of memory. This observation is used when designing the new allocator.

4.7 Custom new and delete

The custom allocator works by combining `malloc` and *LinkedPool*. When a small allocation is processed, *LinkedPool* is used to process the request. When the allocation size is large, `malloc` is used is used to satisfy the request. The name of this allocator is *CustomNew*. Consider the UML diagram of the custom allocator.

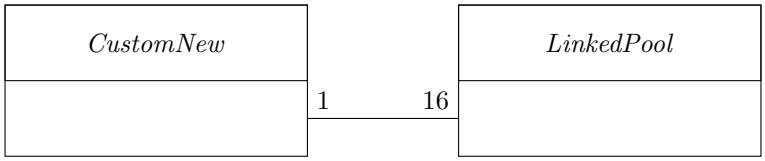

Figure 4.9: UML diagram showing the structure of *CustomNew*.

CustomNew keeps track of 16 *LinkedPool*s. The reason why there are 16 *LinkedPools* associated to *CustomNew* is explained in chapter 7.

In this section, *LinkedPool*s are considered to be typeless. Pools are created for objects of a particular size, as opposed to a particular type. For instance a *LinkedPool* of size 16 denotes that each slot has a size of 16 bytes. The user of the pool can store any type in the slots returned by the allocator as long as the size of the type does not exceed 16 bytes.

The reason why pools are typeless is because `operator new` does not keep track of the type, only the size of the allocation. This operator needs to be able to allocate memory for any size that can fit into a *size_t* (i.e. 64 bits). The main requirement of *CustomNew* is to waste less memory than `malloc`, while performing allocation and deallocation just as quickly. Earlier it was mentioned that *LinkedPool* has large overheads for larger types. Therefore, the first step is to decide for which sizes `malloc` will be used instead of *LinkedPool*.

Listing 4.7: An algorithm for allocating an object using *CustomNew*

```
Algorithm allocate:
  Let size = size of allocation
  if size > threshold:
    use malloc
  else:
    use LinkedPool
```

The `threshold` variable ensures that *LinkedPool* is only used for small allocations where it outperforms `malloc`. The value of the threshold has been determined through observation.

The main design issue is to determine how *LinkedPool* is going to be used when the size is smaller than the value of the threshold. Assume that the value of the threshold is M. One design choice might involve creating M *LinkedPool* allocators. The first allocator will deal with allocations of size 1, the second one with allocations of size 2, and so on until the M^{th} allocator which will be able to allocate objects of size M. This design sounds reasonable, but it has one serious issue. A program might allocate 5 objects of size 8, 800 objects of size 64, and 3 objects of size 16. The *CustomNew* allocator is not going to be able to save any memory. This is because the allocator will allocate 3 pages of memory, one for the slots of size 8, one for slots of size 16, and another one for slots of size 64. In these pages, only a few slots are going to become occupied throughout the execution of the program, which means that a lot of memory will be wasted due to unoccupied slots. These free slots cannot be deallocated because they are part of pools which still contain objects that are in use.

In the general case, it might be the case that some objects are allocated very often, while others are allocated just a few times. Consider another design choice where *CustomNew* allocates $\frac{M}{8}$ *LinkedPool*s instead of M. Each *LinkedPool* will have a size that is a multiple of 8, where first linked pool allocator will have a size of 8, the second one a size of 16, and so on. When an allocation that is smaller or equal to 8 bytes is processed, it will be allocated a slot in the first *LinkedPool*, whose size is 8. This means that all allocations that are smaller than the threshold will be placed into a pool of size S, where S is the size of the allocation rounded to the next multiple of 8.

This design has the same issue as before, but it tries to group similar allocation sizes into a common pool to reduce the number of free slots that are never going to become occupied. Consider the example where a program allocates an integer (size 4), a character (size 1), and a double (size 8). In the first case, the pool allocators for sizes 1, 4 and 8 are all going to allocate a *TypedMemoryPool*. In these memory pools, only 1 slot is going to be occupied, the other 509 will be free[4]. So this program is wasting $509 * 8 = 4072$ bytes per memory pool, which is 12216 bytes in total. The `malloc` function, on the other hand, only wastes 24 bytes. The new design is going to group these 3 allocations into a memory pool of size 8, therefore this particular implementation will only waste 4056 bytes[5].

[4] $\frac{4096-16}{8}$ total slots in pools of size 1, 4 and 8.

[5] 507 free slots, each slot occupies 8 bytes.

This is still worse than `malloc`, but a lot better than before.

Another issue with the new design is that because allocations are grouped, objects that are smaller than the size of the pool are not going to use all the bytes of the slot that was allocated to them. Consider a pool of size 8. Allocating 510 objects of size 1 means that 7 bytes per slot are wasted. `malloc`, on the other hand, wastes 8 bytes per small allocation. The overheads are still smaller, thus the requirement that *CustomNew* should waste less than `malloc` still holds. This design of *CustomNew* should be able to amortize all the overheads based on the following assumption: sizes are usually multiples of the alignment, and most objects have an alignment of 8 bytes, therefore most objects will have a size that is divisible by 8 and will use all the bytes provided by the linked pool allocator. Some allocations are still going to waste bytes, but overall, *CustomNew* should waste less memory than `malloc`.

It is necessary to also provide a deallocation algorithm. In C++, `operator delete` frees the pointer given with `free`. There is no implicit way of knowing if a pointer was allocated through `malloc` or through a *LinkedPool*. The algorithm for deallocation is as follows:

Listing 4.8: A decallotion algorithm for *CustomNew*

```
Algorithm deallocate(ptr):
  Let ptr = the pointer that is deallocated
  if wasMalloced(ptr):
    free(ptr);
  else:
    pool = LinkedPoolOf(ptr);
    pool.deallocate(ptr);
```

In order to implement the `wasMalloced` function, the implementation must keep a list[6] of all pointers that were returned by `malloc`. The reason why these pointers are tracked, and not those allocated through *LinkedPool*, is because large allocations are not as common as small allocations. Tracking only large allocations will result in a list that has a smaller size, thus searching can be performed faster.

The operation `LinkedPoolOf` is described in chapter 5, because it is very implementation specific.

[6] Or other containers that might be suitable.

Chapter 5

Implementation

This chapter describes the implementation of *LinkedPool* and *CustomNew*.

Because *CustomNew* is based on *LinkedPool*, this chapter also describes several optimizations that can make the implementation of the *LinkedPool* allocator perform better.

5.1 LinkedPool

The *LinkedPool* allocator is responsible for managing *MemoryPool*s of a certain type. Since *LinkedPool* is always associated with the type of data it contains, it is implemented as a template class.

Listing 5.1: The LinkedPool class

```
template<typename T>
class LinkedPool {
public:
  void* allocate();
  void deallocate(void* t_ptr);
private:
  std::set<void*> m_freePools;
  std::mutex m_poolLock;
  size_t m_headerPadding;
  size_t m_slotSize;
};
```

Listing 5.1 shows the interface of the *LinkedPool* allocator. This allocator has two methods which are used to allocate and deallocate memory for some type *T*.

5.1.1 Allocation

A general pool allocation algorithm was presented in listing 4.2. Listing 5.2 shows how this high-level description is translated into C++.

Listing 5.2: The implementation of the `allocate` method.

```
1  template<typename T>
2  void* LinkedPool<T>::allocate() {
3    m_poolLock.lock();
4    // look for a pool that has a free slot
5    if (m_freePools.size() != 0) {
6      // the first pool in set must have a free slot
7      return nextFree(*m_freePools.begin());
8    } else {
9      // allocate a new page of memory because there are no free pool
10     // slots left
11     Pool pool = aligned_alloc(getPageSize(), getPageSize());
12     constructPoolHeader(pool);
13     m_freePools.insert(pool);
14     return nextFree(pool);
15   }
16 }
```

The `m_poolLock` member variable is used to ensure thread safety. The `m_freePools` set models the one-to-many relationship between *PoolAllocator* and *TypedMemoryPool*. This set keeps track of all pools that have at least one free slot. Whenever pools become full, they are removed from the set. Please note that *Pool* on line 11 is just an alias for `void*`.

Lines 5-7 handle the case when there is a free slot in one of the memory pools. The `nextFree` method returns a free slot from the pool it receives as an argument.

Lines 11-14 deal with the case when a new pool must be allocated. The `aligned_alloc` function is used to allocate a page of memory that is *page aligned*[1]. The reason why it needs to be page aligned is described in detail in section 5.1.2. Line 12 initializes the new memory pool, by creating the pool header, and the empty slots. After the initialization, the freshly created memory pool is added to the set of pools that have free slots.

constructPoolHeader

This method creates the pool header, and initializes the hidden linked list in the pool it receives as an argument.

Listing 5.3: The implementation of the `constructPoolHeader` method.

```
1  struct PoolHeader {
2    size_t occupiedSlots;
```

[1] Aligned at the start of some page of memory.

```
3     Node head;
4   };
5
6   ...
7
8   template<typename T>
9   void LinkedPool<T>::constructPoolHeader(Pool t_ptr) {
10    auto header = new (t_ptr) PoolHeader();
11    auto first = reinterpret_cast<char*>(header + 1);
12    first += m_headerPadding;
13    header->head.next = reinterpret_cast<Node*>(first);
14    for (size_t i = 0; i < m_poolSize - 1; ++i) {
15      auto node = new (first) Node();
16      first += m_slotSize;
17      node->next = reinterpret_cast<Node*>(first);
18    }
19    new (first) Node();
20  }
```

In listing 5.3, lines 1-4 define the pool header of the linked pool, and lines 9-20 implement the initialization of the pool. Line 10 creates the pool header at the beginning of the pool. Lines 11-13 initialize the head of the linked list by making it point to the first pool slot. At lines 14-19, a *Node* object is created at the beginning of each pool slot, and is added to the linked list.

nextFree

The `nextFree` method returns an empty slot from a pool, and updates the state of the header. When the pool becomes fully occupied, this method also removes the pool from the `m_freePools` set.

Listing 5.4: The implementation of the `nextFree` method.

```
1   template<typename T>
2   void* LinkedPool<T>::nextFree(Pool t_ptr) {
3     auto header = reinterpret_cast<PoolHeader*>(t_ptr);
4     Node& head = header->head;
5     void* toReturn = head.next;
6     if (head.next) {
7       head.next = head.next->next;
8       // if the pool becomes full, don't consider it in the list
9       // of pools that have some free slots
10      if (++(header->occupiedSlots) == m_poolSize) {
11        m_freePools.remove(t_ptr);
12      }
13    }
14    m_poolLock.unlock();
15    return toReturn;
16  }
```

Listing 5.4 shows the implementation of the `nextFree` method. Lines 6-13 remove the free slot from the linked list, because it is no longer available. The number of occupied slots is increased by one, which means care must be taken to ensure that the pool is no longer considered to be part of the `m_freePools` set, if it becomes fully occupied. Line 14 ensures the lock is released, because the lock is always acquired before `nextFree` is called. For instance, `allocate` acquires the lock before calling `nextFree`.

5.1.2 Deallocation

Before translating listing 4.3 into C++ code, it is important to describe how the `poolOf` function is implemented. A potential design of this method was described in section 4.2. The design has a worst-case running time of $O(n)$, where n is the number of *TypedMemoryPools* allocated by *PoolAllocator*. In order for this implementation to match the speed of `malloc`, this operation would need to happen in constant time. In section 5.1.1, it was specified that pools are page aligned. This means that the start address of any given pool has its first 3 low-order hexadecimal digits equal to 0. Consider a pool whose start address is 0x12345000. Every slot of this pool will have an address in the range $[0x12345000, 0x12346000)$. This means that, to obtain the pool a particular pointer belongs to, the first 3 low-order hexadecimal digits of the pointer need to be set to 0. The new address will always represent the pool of the given pointer, if and only if the pointer was allocated with *LinkedPool*.

Listing 5.5 shows the implementation of the pool deallocation algorithm.

Listing 5.5: The implementation of the `deallocate` method.

```
1  template<typename T>
2  void LinkedPool<T>::deallocate(void* t_ptr) {
3    // get the pool of t_ptr
4    auto pool = reinterpret_cast<PoolHeader*>(
5      reinterpret_cast<size_t>(t_ptr) & getPoolMask()
6    );
7    m_poolLock.lock();
8    // the last slot was deallocated => free the page
9    if (pool->occupiedSlots == 1) {
10     m_freePools.remove(pool);
11     free(pool);
12   } else {
13     auto newNodeG = new (t_ptr) Node();
14     // update nodes to point to the newly create Node
15     Node& head = pool->head;
16     newNodeG->next = head.next;
17     head.next = newNodeG;
18     // the pool is not full, therefore add it to the list of pools
19     // that have free slots
20     if (--pool->occupiedSlots == m_poolSize - 1) {
21       m_freePools.insert(pool);
```

```
22        }
23      }
24      m_poolLock.unlock();
25    }
```

Lines 4-6 retrieve the pool of the given pointer in constant time. If all the slots in the pool become free, the memory block that was allocated with `aligned_alloc` is freed, and the pool is removed from the set. This is implemented at lines 9-11. At lines 13-22, a new *Node* is created at the beginning of the slot that is being deallocated, and it is inserted after the head of the linked list. If a pool is fully occupied, and one of its slots becomes free, the implementation adds the pool back into the set of pools that have free slots.

5.2 Improvements and optimizations

5.2.1 avl_tree

In C++ *std::set* is implemented using Red Black Trees [11]. There is a lot of debate on whether Red Black Trees are better than AVL trees or not [12, 13]. To decide which implementation performs better, it is necessary to benchmark two different *LinkedPool* implementations, one of which uses *std::set*, and another one which uses a highly performant AVL tree implementation called *avl_tree*.

5.2.2 light_lock and LMLock

An external library provides the *light_lock* class which is a very fast mutex implementation for X86 systems. This mutex is a lot faster[2] than *std::mutex*. In C++ it is possible to check the architecture of the system at compile time using the C preprocessor. The *LMLock* class was created as part of this project in order to ensure that the allocators are thread safe on any system. If a program that uses *LinkedPool* is compiled for an architecture other than X86, it should still remain thread safe. This is done by falling back to *std::mutex* in those cases. *LMLock* provides a mutex that on X86 systems uses *light_lock*, and on other systems uses *std::mutex*. The locking mechanism is decided at compile-time based on the architecture of the target system.

Listing 5.6: The interface of *LMLock*.

```
class LMLock {
public:
  void lock();
  void unlock();
private:
#ifdef __x86_64
  light_lock_t m_lock;
#else
```

[2]The speed-up can be seen in the running time of the benchmarks.

```
  std::mutex m_lock;
#endif
};
```

Listing 5.6 shows the interface of *LMLock*. This class can be used with constructs such as *std::unique_lock*, because it implements the interface of a lock. The implementation can be found in the appendix in listing B.44.

5.2.3 Caching pools

Whenever a slot is allocated or deallocated from a pool P, P is cached, and saved into a member variable. On subsequent allocations, P will be used to satisfy the requests.

This optimization is based on the fact that accessing pools may cause page faults. Page faults are very slow, and if these happen frequently, the programs are slowed down considerably. When there is page fault, the page is loaded into main memory, but another page, called the victim, is 'evicted'. The OS is responsible for choosing the victim page. Usually, the OS chooses the least recently used page as a victim. This means that when a page fault happens, the recently retrieved page has a lower chance of being evicted. Because pools might cause page faults, it is a good idea to cache pools that have recently been accessed. This ensures that when another allocation request is processed, the slot will be allocated in one of the cached pools which is known to be in memory, and not on disk. Allocating slots from other pools could cause more page faults.

5.2.4 Benchmarks

Tables 5.1, and 5.2 show how the running time improves after implementing the three optimizations mentioned above. All benchmarks allocate 1,000,000 objects of size 24, and record the allocation and deallocation times of *LinkedPool*.

Benchmark	LinkedPool	+ avl_tree	+ LMPool	+ caching
normal	307.77	309.27	294.92	196.52
specified	167.73	170.74	124.97	116.74
random	307.54	309.45	294.97	197.70
random2	439.96	461.07	356.35	333.59
worst	308.73	307.20	300.66	222.84

Table 5.1: Allocation times of all implementations in ms.

Benchmark	LinkedPool	+ avl_tree	+ LMPool	+ caching
normal	307.14	296.35	196.06	194.14
specified	159.59	159.11	104.93	106.15
random	2544.70	2556.07	1718.47	1657.32
random2	4093.86	4107.36	3749.97	3741.15
worst	1511.24	1484.34	1572.57	1578.27

Table 5.2: Deallocation times of all implementations in ms.

Tables 5.1 and 5.2 show that the difference between *std::set*, and *avl_tree* is negligible.

LMLock speeds up deallocation considerably. It also improves allocation speed, but not by a large margin.

Caching on the other hand, when combined with the other two optimizations, provides a great overall speed-up. The reason why *LinkedPool* uses *avl_tree* is because it provides easy access to the root of the tree in constant time, while *std::set* does not provide this operation.

5.3 Custom new

The *CustomNew* allocator provides two functions, namely `custom_new` and `custom_delete`, which are used to replace `operator new` and `operator delete`.

Listing 5.7: The interface of the *CustomNew* allocator.

```
void* custom_new_no_throw(size_t t_size,
                          size_t t_alignment=alignof(max_align_t));

void* custom_new(size_t t_size,
                 size_t t_alignment=alignof(max_align_t));

void custom_delete(void* t_ptr) noexcept;
```

Listing 5.7 shows the declarations of the functions that are used to replace `new` and `delete`. In C++, `operator new` throws *std::bad_alloc* when an allocation fails. The `operator new` function can also return a `nullptr` when allocation fails, if the user provides the *std::no_throw* argument to `new`. This is the reason why the interface provides two `custom_new` functions.

Before showing the implementation details, it is important to define some classes that will help implement the specifications of this allocator.

In section 4.7, it was mentioned that pools are considered typeless. *LinkedPool* is a template class, so it is not typeless. Therefore, it is necessary to create another allocator, *GlobalLinkedPool*, which is going to satisfy this requirement. The two main differences between *LinkedPool* and *GlobalLinkedPool* are that the latter has a constructor which takes the desired size and alignment as pa-

rameters, and that it records extra information in the pool header. The former uses the `sizeof` and `alignof` operators on the template parameter to get the size and the alignment of the type. The header of each pool created by *GlobalLinkedPool* also keeps track of the slot size, thus making the header 24 bytes long.

In order to model the one-to-many relationship from figure 4.9, a new class called *GlobalPools* is introduced. This class keeps track of 16 *GlobalLinkedPool*s, and its interface is given in listing 5.8.

Listing 5.8: The interface of the *GlobalPools* class.

```
1  class GlobalPools {
2    public:
3      GlobalPools(size_t t_numOfPools);
4      GlobalLinkedPool& getPool(size_t t_size);
5    private:
6      std::vector<rpools::GlobalLinkedPool,
7                  mallocator<rpools::GlobalLinkedPool>
8      > m_pools;
9  };
```

The `getPool` method returns the pool that can hold the specified size. This method requires the argument to be a multiple of 8, because all the *GlobalLinkedPools* of the class can hold sizes up to a multiple of 8. Please note that pools that have a size that is a multiple of 8 have a slot alignment of 8 bytes, while pools of size 16, have 16 bytes aligned slots.

The *std::vector* class can take an optional template argument which specifies the type of allocator it should use. In this case, `m_pools` uses *mallocator* to allocate space, as seen at line 7. This allocator is used to ensure that *std::vector* does not call `new` or `delete`, but uses `malloc` and `free` instead. The `operator new` function is overriden, and calls `custom_new`. If `custom_new` creates *std::vector*s, which in turn call *operator new*, then this will cause a stack overflow. In the case of *GlobalPools*, *mallocator* is used to prevent this.

Listing 5.9: The implementation of the *CustomNew* allocator.

```
1  namespace {
2    using namespace rpools;
3
4    const size_t __threshold = 128; // malloc performs equally well
5                                    // on objects of size > 128
6    const size_t __mod = sizeof(void*) - 1;
7    const size_t __logOfVoid = std::log2(sizeof(void*));
8
9    // Used to mark the first 16 bytes of a malloc-d region
10   struct MallocHeader {
11       char validity[16] = "          \0";
12   };
13
```

```
14      GlobalPools& getPools() {
15          static GlobalPools pools(__threshold >> __logOfVoid);
16          return pools;
17      }
18  }
19
20  void* custom_new_no_throw(size_t t_size, size_t t_alignment) {
21      if (t_size > __threshold) {
22          auto addr = static_cast<char*>(std::malloc(t_size +
23                                                     sizeof(MallocHeader)));
24          auto header = new(addr) MallocHeader();
25          std::strcpy(header->validity, "IsThIsMaLlOcD!\0");
26          return addr + sizeof(MallocHeader);
27      } else {
28          size_t remainder = t_size & __mod; // t_size % sizeof(void*)
29          // round up to the next multiple of sizeof(void*)
30          t_size = remainder == 0 ? t_size : (t_size + __mod) & ~__mod;
31          t_size += (mod(t_size, t_alignment)) == 0 ? 0 : sizeof(void*);
32          void* addr = getPools().getPool(t_size).allocate();
33          return addr;
34      }
35  }
36
37  void* custom_new(size_t t_size, size_t t_alignment) {
38      void* toRet = custom_new_no_throw(t_size, t_alignment);
39      if (toRet == nullptr) {
40          throw std::bad_alloc();
41      }
42      return toRet;
43  }
44
45  void custom_delete(void* t_ptr) noexcept {
46      auto cAddr = reinterpret_cast<char*>(t_ptr);
47      cAddr -= sizeof(MallocHeader);
48      auto header = reinterpret_cast<MallocHeader*>(cAddr);
49      if (std::strcmp(header->validity, "IsThIsMaLlOcD!\0") == 0) {
50          free(cAddr);
51      } else {
52          const PoolHeaderG& ph = GlobalLinkedPool::getPoolHeader(t_ptr);
53          getPools().getPool(ph.sizeOfSlot).deallocate(t_ptr);
54      }
55  }
```

The system must be able to differentiate between pointers that have been returned by `malloc`, and those that have been returned by *GlobalLinkedPool*. Lines 10-12 define a structure which will be used to mark pointers that are returned by `malloc`. Lines 21-27 describe the case when `malloc` is used to satisfy the request. Instead of allocating `t_size` bytes, the implementation allocates

`t_size + 16` bytes to store an additional *MallocHeader*[3]. This overhead is hidden from the user by adding 16 bytes to the address that is returned. This ensures that *MallocHeader* is not overwritten. Lines 46-49 implement the check if the pointer was allocated with `malloc` or not. This is done by subtracting 16 bytes from the pointer, and checking whether the first 16 bytes can be correctly interpreted as a string which indicates that the pointer was indeed created through `malloc`. If the specific string is not found, then it must be the case that the pointer belongs to a *GlobalLinkedPool*[4]. This method has one obvious flaw. There is a chance that it might detect a pointer as being allocated with `malloc` when it is not. However, there is a 1 in $2^{64} - 1$ chance of this to happening, which is very low.

An alternative approach is to store all pointers returned by `malloc` in a container. This approach is poor, because the pointer check would take $O(n)$ time, where n is the number of pointers allocated with `malloc`. The *MallocHeader* approach performs this check in constant time, which is more desirable. Please note that both methods will use additional memory to store the information whether a pointer was returned by `malloc` or not. The two approaches use the same amount of memory to store this information, but their running times are different.

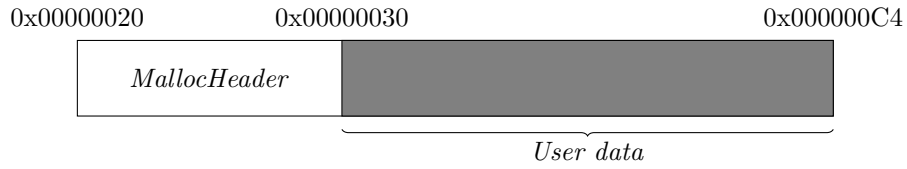

Figure 5.1: A marked `malloc` pointer.

For example, if a user allocates 148 bytes, `custom_new` will use `malloc` to satisfy the request. The pointer returned by `malloc` is going to be marked, as seen in figure 5.1. Suppose that `malloc` returns the pointer 0x00000020. The user of `custom_new` will receive the pointer 0x00000030, because the implementation hides the *MallocHeader*. If the user deallocates the pointer 0x00000030, then the *MallocHeader* can be found by subtracting 16 from this address, as seen in figure 5.1.

The `__threshold` value is 128 because *GlobalLinkedPool* allocation and deallocation times decrease for objects of larger sizes. This was observed during testing and benchmarking.

Lines 28-33 use *GlobalLinkedPool* to satisfy the requests smaller than 128 bytes. This is done by rounding the given size to the next multiple of 8 or 16 depending on the given alignment. It is necessary to round the size because of the `getPool` method of the *GlobalPools* class. Pools of sizes that are divisble by

[3]Because `sizeof(MallocHeader)` = 16.

[4]C++ does not have a `byte` type, but a `char` has a size of 1 byte. The string denotes just a sequence of bytes, and the actual sequence is hard-coded into the implementation.

16 can satisfy any alignment as long as it is not over 16. Thus, if the requested alignment is 8 or less, the size is rounded to the next multiple of 8. The allocated size is rounded to a multiple of 16, if and only if the required alignment is 16.

In order to speed up the implementation, instead of using the regular modulo operator, it is possible to use the bitwise-and operator to compute the modulo of a number. Bitwise-and can be used to calculate the modulo, if and only if the right hand side of the modulo operation is a power of two. In the case of alignments this is always the case, thus the implementation can use bitwise-and to speed up the execution. This is seen at line 28, and 31 where the `mod` function is responsible for carrying out this optimization.

If a pointer is deallocated, and it was not returned by `malloc`, then it was surely allocated with a *GlobalLinkedPool*. The implementation has to find out which of the 16 pools returned this pointer. Earlier it was mentioned that the new pool header also keeps track of the size of the slot. This means that given a pointer that is known to be allocated by a pool, the address can be masked in order to get the pool header, and from there the size of the slot. *GlobalPools* offers a method for retrieving the pool which has a certain size. The implementation calls this method, and passes the size of the slot as a parameter. The returned *GlobalLinkedPool* can be use to safely deallocate the pointer.

5.4 Overriding and injecting CustomNew

This section describes two different methods of making *CustomNew* available to any program without changing the source code.

LinkedPool and *CustomNew* can be used directly in any program, but as it was described in chapter 3, the user must also be able to inject these into programs without needing to modify their source files. In this chapter, *linkedpool* is going to be used to refer to the shared library which contains the implementation of *LinkedPool*. Similarly, *customnew* is a shared library which contains the implementation of *CustomNew*, but also the new implementations of `operator new` and `operator delete`.

5.4.1 The LD_PRELOAD trick

When *customnew* is linked with a program, there will be two definitions for `operator new` and `operator delete`, one provided by the standard C++ library, and one provided by *customnew*. The linker will use the first definition it finds to 'implement' `operator new` and `operator delete`. The C and C++ standard libraries are the first libraries that are linked to a program. This means that if *customnew* is linked after these, the `new` and `delete` operators will use the C++ implementations of `operator new` and `operator delete`.

The *LD_PRELOAD* environment variable is used to instruct the linker to load the specified shared libraries *before* any other libraries. This means that

setting the *LD_PRELOAD* variable to the path of *customnew* will enable programs to use *CustomNew* without changing their source code. If the program uses the `new` operator, then the `custom_new` function will be called instead[5].

The main advantage of this method is that *customnew* can be injected into any binary. It does not require any source code modifications, and users can see the results of using a memory pool very easily. However, the main disadvantage is that all objects will be 16 bytes aligned. The `operator new` function always aligns objects at 16 bytes boundaries. Because of this, `operator new` does not have access to the alignment of the allocation. Therefore, `custom_new` will be asked to align objects at 16 bytes boundaries, which will cause the program to waste memory.

5.4.2 The LLVM method

The *CustomNewPass* is a basic block pass that changes all the calls to `operator new` and `operator delete` with calls to `custom_new` and `custom_delete`. This is done by looping through all instructions of a basic block, and modifying all *CallInst* and *InvokeInst* instances whose first argument is either `operator new` or `operator delete`. Listing 5.11 shows the LLVM bytecode generated from the C++ program shown in listing 5.10.

Listing 5.10: A C++ program which uses `new` and `delete`.

```
1  int main() {
2    int* x = new int();
3    delete x;
4    return 0;
5  }
```

Listing 5.11: An LLVM function which calls `new` and then `delete`.

```
1  ; Function Attrs: noinline norecurse optnone uwtable
2  define i32 @main() #0 {
3    ...
4    %3 = call i8* @_Znwm(i64 4) #3
5    %4 = bitcast i8* %3 to i32*
6    store i32 0, i32* %4, align 4
7    store i32* %4, i32** %2, align 8
8    %5 = load i32*, i32** %2, align 8
9    %6 = icmp eq i32* %5, null
10   br i1 %6, label %9, label %7
11
12   ; <label>:7:                                     ; preds = %0
13   %8 = bitcast i32* %5 to i8*
14   call void @_ZdlPv(i8* %8) #4
15   br label %9
```

[5]This happens because `operator new` was overriden, and now it calls `custom_new` instead of `malloc`

```
16
17      ; <label>:9:                                      ; preds = %7, %0
18      ret i32 0
19    }
```

Line 6 of listing 5.11 is a call to `new`[6], and can be translated to `call void* operator new(i64 4)`. Line 16 is a call to delete (`call void operator delete(i8* %8)`).

These two lines represent the calls that need to be changed. At line 7, there is a *bitcast* from i8* to i32*, that is preceded by the call to `new`[7]. Its role is to convert an i8* to an i32*. In this case, the i8* that is converted is the return value of the call to `new`. This means that the next instruction after the call to `new` can be used to find out the type that is allocated. This is important because if the type is known, so is the alignment. The `custom_new` function takes an extra argument which is the alignment. Alignments that are greater than 8 cause objects to be placed into pools where slots are 16 bytes aligned. This causes programs to lose a lot of memory. Using the LLVM approach ensures that the correct alignment is passed to `custom_new`, which means the allocation will be placed into a more suitable pool which reduces the memory usage. The alignment is easy to extract for base types, and for base classes. However, it was necessary to come up with a recursive solution which finds the alignment of complex structures such as classes that inherit from any number of classes.

The allocated type is easy to find in the case when there is a call to `new`. However, `new` can also be invoked, rather than called.

Listing 5.12: Basic blocks which contain an invoke to `new`.

```
1   ; <label>:54:                                     ; preds = %52
2
3   ...
4
5   %56 = invoke i8* @_Znwm(i64 48) #15
6           to label %57 unwind label %69
7
8 ; <label>:57:                                     ; preds = %54
9   %58 = bitcast i8* %56 to %class.Sudoku*
10
11  ...
```

The *invoke* instruction calls the specified function, `new` in this case. If the function returns normally, then execution jumps to basic block *%57*, but if an exception occurs, then the execution jumps to basic block *%69*. This information is specified on line 6 of listing 5.12. Note that there is no *bitcast* instruction after the invoke to `new`. It has been observed that the *bitcast* instruction that holds the type of the allocation, can be found in basic block *%57*, which is the basic

[6] *_Znwm* is the mangled version of `operator new`.
[7] The i8* type is actually void* in C++.

block to which execution jumps when the invoke to `new` returns successfully. This means that the type can be extracted from the first *bitcast* instruction found in the 'destination' basic block of the *invoke* instruction.

The implementation of this pass can be found in the appendix in listing B.58.

The main disadvantage of this method is that in order to inject *CustomNew* into programs, they have to be compiled with clang, and the pass must be run as part of the compilation process by specifying an extra argument to the compiler.

5.5 CustomNewDebug

This is an allocator that is injected into programs through LLVM, and keeps track of the number of objects that have been allocated, and deallocated.

Listing 5.13 shows the implementation of the *CustomNewDebug* allocator.

Listing 5.13: The implementation of *CustomNewDebug*

```
namespace {
  AllocCollector ac;
}

void* custom_new_no_throw(size_t t_size, size_t t_alignment,
                          const char* t_name, size_t t_baseSize,
                          const char* t_funcName) {
  void* toRet = malloc(t_size);
  ac.addObject(t_size, t_alignment, t_name, t_baseSize, t_funcName,
      toRet);
  return toRet;
}

void* custom_new(size_t t_size, size_t t_alignment,
                 const char* t_name, size_t t_baseSize,
                 const char* t_funcName) {
  void* toRet = custom_new_no_throw(t_size, t_alignment, t_name,
  t_baseSize, t_funcName);
  if (toRet == nullptr) {
    throw std::bad_alloc();
  }
  return toRet;
}

void custom_delete(void* t_ptr) noexcept {
  ac.removeObject(t_ptr);
  free(t_ptr);
}
```

In this case, `custom_new` takes additional arguments. These arguments will be computed at compile time using LLVM. The *CustomNewDebugPass* is a basic block pass that works very similarly to *CustomNewPass*. This pass supplies

extra information to the debug version of `custom_new`, such as the name of the function that calls `new`, the name of the type, and the base size of the allocated type. A program might allocate an array of 10 integers, in which case the size of the allocation is 40 bytes, but the base size of the allocation is 4 bytes. The implementation of this pass can be found in the appendix in listing B.59.

The *AllocCollector* class is responsible for recording all the information that is given to `custom_new`. This is done by saving the information in a map. Note that the contents of the map change every millisecond, thus the implementation needs to spawn a thread which blocks execution every 10ms to save the state of the map into a data structure which is written to a file when execution ends. The problem with this approach is that the snapshots are kept in memory, and therefore, the program might run out of memory quickly. Listing 5.14 shows the interface of the *AllocCollector* class.

Listing 5.14: The interface of the *AllocCollector* class.

```
class AllocCollector {
  public:
    AllocCollector();
    void addObject(size_t t_size, size_t t_align,
                   const char* t_name, size_t t_baseSize,
                   const char* t_funcName, void* t_ptr);
    void removeObject(void* t_ptr);
    void takeSnapshot();
    virtual ~AllocCollector();
  private:
    ...
    json m_snapshots;
    ...
    size_t m_snapshotCount = 0;

    void run();
};
```

The `addObject` method, shown in listing 5.14, records the given allocation in the map. The last parameter of this method represents the pointer to the object that is being allocated. This is used as the key in the map of allocations. The `removeObject` method removes an entry from the map. The reason why pointers are the keys is because `custom_delete` has only access to the pointer being deleted. The `takeSnapshot` method is called every 10ms, and saves the information stored in the map into `m_snapshots`. When the destructor of *AllocCollector* is called, the class writes the contents of `m_snapshots` into the file specified by `m_objectsFile`.

A possible output of a program that was compiled with the *CustomNewDebugPass* is shown in listing 5.15.

Listing 5.15: An example of one snapshot.

```
1 {
```

```
2  [
3    {
4      "Sudoku": {
5        "8": {
6          "48": {
7            "array": 1,
8            "base_size": 48,
9            "current": 4,
10           "function": "Sudoku::successors() const",
11           "peak": 4
12         }
13       }
14     },
15   "std::pair<int, int>": {
16     "4": {
17       "128": {
18         "array": 16,
19         "base_size": 8,
20         "current": 0,
21         "function": "__gnu_cxx::new_allocator<std::pair<int, int>
               >::allocate(unsigned long, void const*)",
22         "peak": 1
23       },
24       "16": {
25         "array": 2,
26         "base_size": 8,
27         "current": 0,
28         "function": "__gnu_cxx::new_allocator<std::pair<int, int>
               >::allocate(unsigned long, void const*)",
29         "peak": 1
30         }
31       }
32     }
33   }
34 ]
```

Listing 5.15 is an example of a snapshot in JSON format. At lines 4-14, it can be seen that there are some *Sudoku* objects in memory. The *Sudoku* type is 8 bytes aligned, and it has an allocation size of 48 bytes. The base size is also 48 bytes, which means that the user always allocates single Sudoku objects, and not arrays. At the moment the snapshot was taken, there were 4 allocations of this type in memory, and the peak number of allocations was 4. These allocations were made in the function *Sudoku::successors() const*.

The *std::pair⟨int,int⟩* type has two entries in the file. The first entry (line 17) suggests that an allocation of 128 bytes was made, but the base size is 8 bytes. This means that the program allocated 16 contiguous objects of this type. Similarly for line 24, where there are only 2 contiguous objects of type *std::pair⟨int,int⟩*.

For larger programs the amount of information that is generated increases substantially. Manually examining JSON files to determine the memory usage of the program is very tedious. The format of a snapshot is not very readable to users. All these issues can be solved by creating a program that can show this data in a more sensible format.

The Python script *generate_obj_alloc_html.py* is responsible for creating an HTML webpage that will provide the user with the utilities mentioned above. Figure 5.2 shows an example of a page that is generated from a JSON file.

Filter types: Sudoku **Filter functions:**

Snapshot	Type Name	Alignment	Size of allocation	Currently allocated	Peak	Allocated in
1	Sudoku	8	48	9	11	Sudoku::successors() const
2	Sudoku	8	48	9	12	Sudoku::successors() const
3	Sudoku	8	48	0	14	Sudoku::successors() const

Figure 5.2: The *Sudoku* objects allocated during the execution of a program.

Figure 5.2 shows that the rendered page provides a filter functionality for both types and functions. This is useful in cases when the user wants to check the number of allocated objects of a certain type, or find out if a certain function allocates too many objects. Note that if a user clicks any of the headers of the table, the table will be sorted according to the clicked header.

5.6 Testing

All of the tools and libraries have been unit-tested, except for the LLVM passes. This is due to the fact that it is not trivial to test an LLVM pass. The best way to test the compiler passes is to run them on complex programs. If a program compiles successfully, and it runs as expected, then there is a high chance that the pass works correctly. The passes have been tested on several non-trivial programs, such as a game, and the Popf planner.

Testing the allocators has been a challenging task. There are test cases which check whether the returned pointers are correctly aligned, or if the *PoolHeader* contains the data that is expected. The best test case is to run the allocator on a program. If programs compiled with the *CustomNewPass* work as excepted, then the pools work correctly as well, because *CustomNew* uses *GlobalLinkedPools*.

Most bugs have been found by running *CustomNewPass* on the Popf planner. This program uses a lot of memory, and also allocates complex structures. This means that small bugs in the implementation crash the program very easily. Some bugs were trivial to fix, while others required some research.

Chapter 6

Professional issues

The code of conduct of the British Computing Sociecty (BCS) was taken into account while developing this project.

In order to implement some of the features and optimzations of the allocators presented in this report, it was necessary to use a few external open-source libraries. For instance unit testing was done with the Catch2 testing framework. The *LinkedPool* allocator relies on the performance of *avl_tree*, and *light_lock*, both of which are open-source libraries. The authors of these libraries are acklowledged in the appendix in the file B.3. These libraries reduced the development time of the project considerably, and becaues of the extra time, it was possible to create several injection mechanisms.

This project will be open-sourced in order to allow other programmers to contribute, and use the allocators in their own software. The project is heavily documented in order to make contributing to the project easier.

The advantages and disadvantages of the allocators presented in this report are well documented, and the performance is described in great detail in chapter 7.

Tools such as GDB, Valgrind, and Clang-Tidy have been used to ensure that the project meets certain quality standards. Valgrind and Clang-Tidy report no errors or warnings. The project was developed such that it works on multiple platforms.

Chapter 7

Benchmarking

There are many benchmarks that have been implemented in order to assess the performance of the *LinkedPool* allocator. *LinkedPool* will be compared to two of the best and most popular pool implementations: *boost* and *MemoryPool*, and to `malloc`. When running benchmarks, the most optimized version of *LinkedPool* will be used. Please note that benchmark results do not include *boost*. This is because deallocation times are very high, and therefore it is infeasible to run the benchmarks. The deallocation cannot be skipped because *boost* crashes in the case where the objects are not deallocated. Thus, it was easier to remove it from the benchmarks. *LinkedPool* is comparable to *boost_pool* when allocating small objects, but *boost* is very slow when deallocating memory.

7.1 Allocation and deallocation time benchmarks

There are several benchmarks that record the time taken to allocate and deallocate objects of a certain type. All of them differ in the order in which objects are allocated and deallocated. All benchmarks take as input the number of objects to be allocated. The objects that are allocated have a size of 24 bytes.

7.1.1 The plot_elapsed_time script

This is a python script which runs a given benchmark several times on inputs of different size, and plots the results using *matplotlib*.

Listing 7.1: Sample execution of the plot_elapsed_time script.

```
python3 plot_elapsed_time -b bench_worst -n 100000
```

In listing 7.1 the script is run on the *bench_worst* benchmark. The script is going to run this benchmark with an input of 10,000 objects, then 20,000 objects, and so on until 100,000. When a benchmark is complete, the script

records the output, and saves it for later use. When all benchmarks are done, the data is plotted.

7.1.2 bench_normal

This is a benchmark which allocates N objects, and then deallocates them in reverse order. This is a rather simple benchmark, but it is important to see how *LinkedPool* compares with other allocators in simpler cases as well. Table 7.1 shows the allocation and deallocation times of the three allocators.

Operation	LinkedPool	MemoryPool	new/delete
allocation	277.76	101.91	428.73
deallocation	193.59	111.46	166.02

Table 7.1: Allocation and deallocation times for the three allocators on *bench_normal* when allocating 10,000,000 objects.

It can be seen that *LinkedPool* performs better than `new`, but worse than *MemoryPool*. *MemoryPool* performs allocations almost 4 times as fast as `new`, while *LinkedPool* performs only twice as fast. Figure 7.2 shows that *LinkedPool* underperforms when objects are deallocated in a sequence.

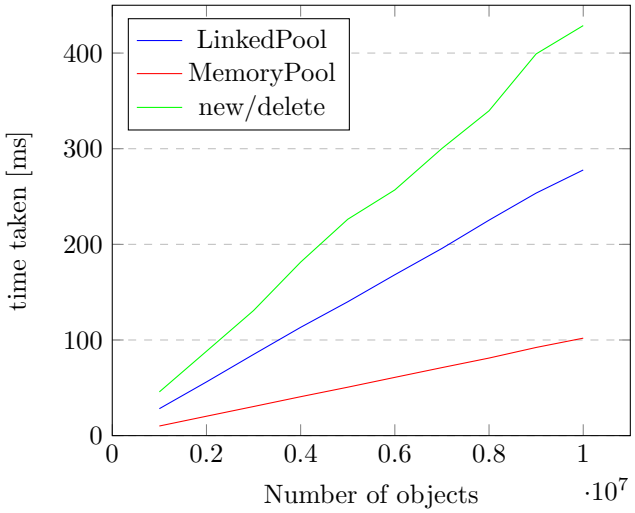

Figure 7.1: Allocation times of *bench_normal*.

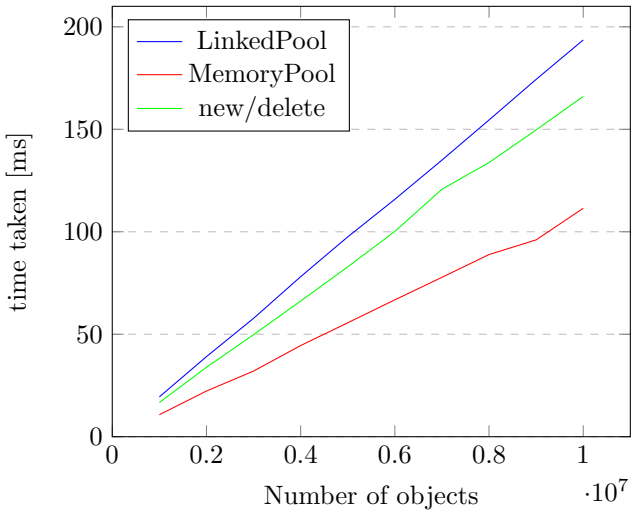

Figure 7.2: Deallocation times of *bench_normal*.

7.1.3 bench_specified

This is a benchmark which allocates and deallocates a number objects in a certain predefined order. The actual order can be found in listing B.69. This is also a simpler benchmark, but this time it simulates a program that allocates a smaller number of objects, and then deallocates some of them. For instance a game might have this allocation pattern. For each frame, a number of objects are allocated, and by the time a new frame is rendered, some of them are deallocated.

Operation	LinkedPool	MemoryPool	new/delete
allocation	118.37	63.91	149.83
deallocation	110.18	58.03	91.46

Table 7.2: Allocation and deallocation times for the three allocators on *bench_specified* when allocating 10,000,000 objects.

The result is similar to *bench_normal*. *MemoryPool* outperforms both allocators, and *LinkedPool* allocates faster than `new`, but deallocates slower than `delete`.

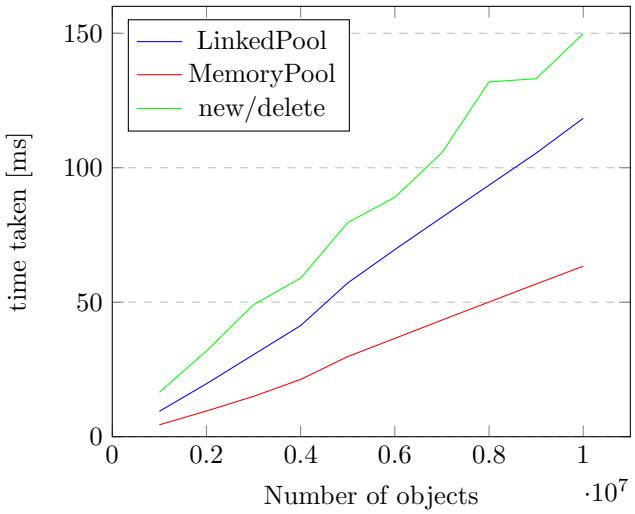

Figure 7.3: Allocation times of *bench_specified*.

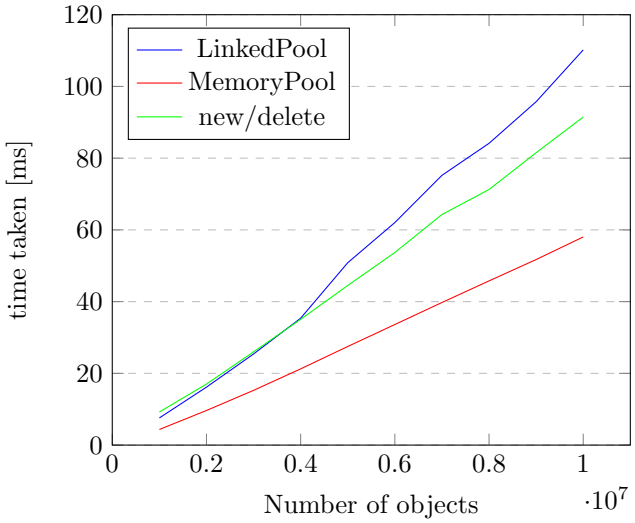

Figure 7.4: Deallocation times of *bench_specified*.

7.1.4 bench_random

This benchmark allocates a large number of objects, and deallocates them in a random order. Allocations are stored in a *std::vector*. This means that the algorithm generates a random list of indices which specifies the random order in which the objects should be deallocated. This list is shared between the three

allocators to ensure that all of them use the same deallocation sequence.

Operation	LinkedPool	MemoryPool	new/delete
allocation	282.59	101.55	412.50
deallocation	1639.80	1378.75	2454.75

Table 7.3: Allocation and deallocation times for the three allocators on *bench_random* when allocating 10,000,000 objects.

Both table 7.3 and figure 7.5 show that *LinkedPool* is now somewhere between the other two allocators in terms of allocation speed. Figure 7.6 shows that *LinkedPool* is now closer to *MemoryPool* in terms of deallocation speed.

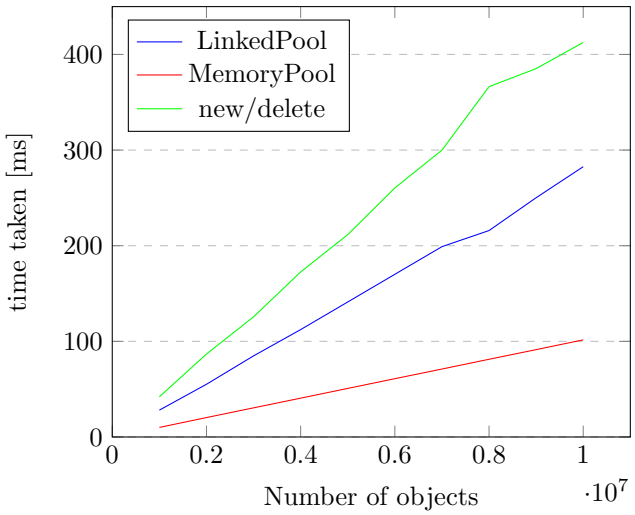

Figure 7.5: Allocation times of *bench_random*.

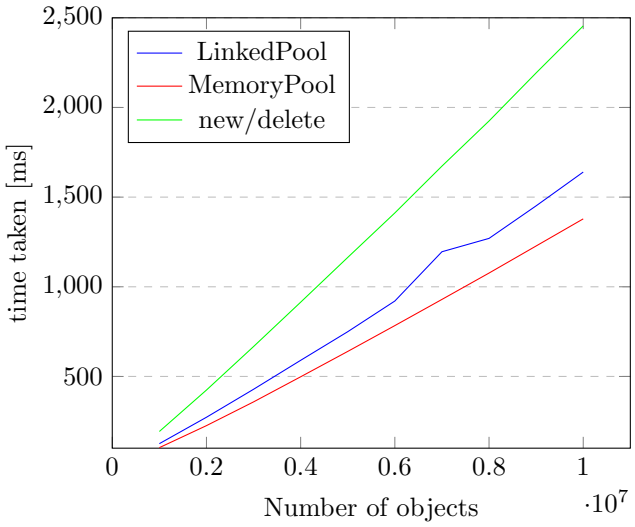

Figure 7.6: Deallocation times of *bench_random*.

7.1.5 bench_random2

This is a another random benchmark which works similarly to *bench_random*. It involves allocating a random number of objects, and then deallocating random objects from the list of allocated objects. The main difference between the two is the fact that this benchmark does not allocate all objects, but only a random number of them. This is a benchmark that generates a random pattern of memory allocations, and deallocations which mirrors the pattern of most programs.

Operation	LinkedPool	MemoryPool	new/delete
allocation	30.00	25.72	65.76
deallocation	302.37	273.85	329.65

Table 7.4: Allocation and deallocation times for the three allocators on *bench_random2* when allocating 1,000,000 objects.

In the case of *bench_random2*, it can be seen in figure 7.7 that *LinkedPool* is closer to *MemoryPool* than it is to `new`. Figure 7.8 shows that the 3 allocators have the same growth, but *MemoryPool* is still faster than *LinkedPool*.

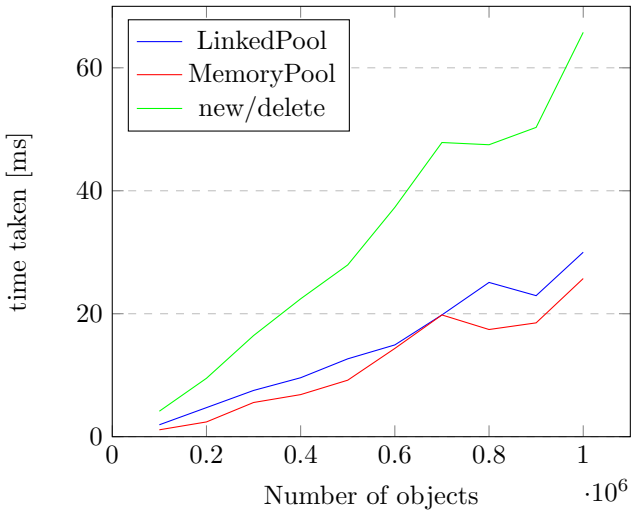

Figure 7.7: Allocation times of *bench_random2*.

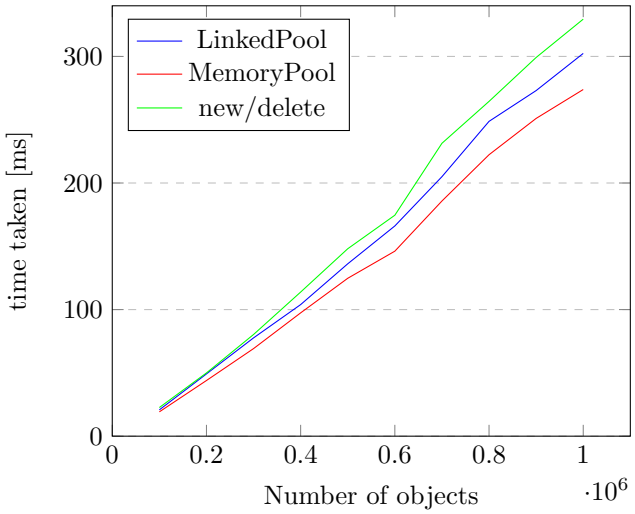

Figure 7.8: Deallocation times of *bench_random2*.

7.1.6 bench_worst

So far all benchmarks were based on possible allocation patterns that programs might use. However, *bench_worst* is a benchmark which generates an allocation, and deallocation sequence which tries to make *LinkedPool* underperform. This is done by generating as many page faults as possible, and by always deallocating objects that are known to be in different pools.

Operation	LinkedPool	MemoryPool	new/delete
allocation	312.11	102.45	429.95
deallocation	1525.69	966.04	1946.30

Table 7.5: Allocation and deallocation times for the three allocators on *bench_worst* when allocating 10,000,000 objects.

Both figures 7.9 and 7.10 show that even though this benchmark allocates, and deallocates objects in such a way that makes *LinkedPool* underperform, the pool allocator is still faster than `new` and `delete`.

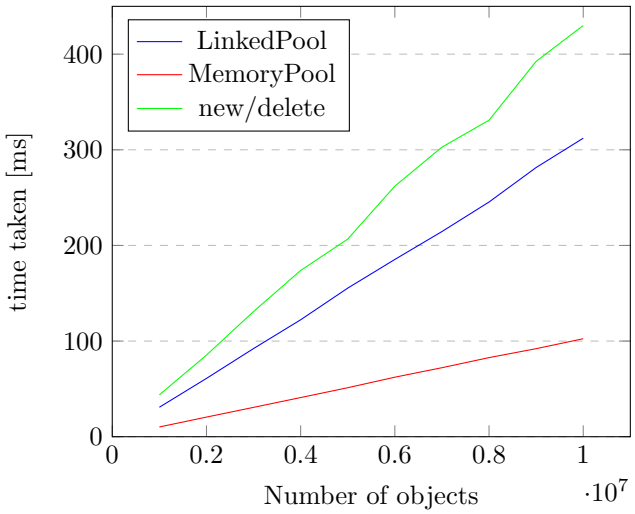

Figure 7.9: Allocation times of *bench_worst*.

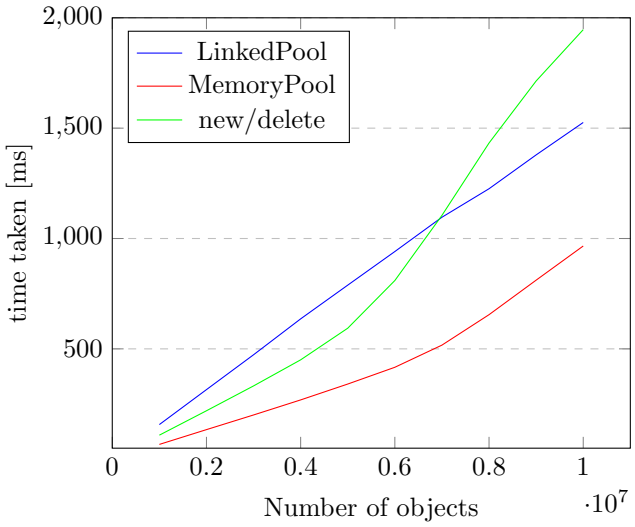

Figure 7.10: Deallocation times of *bench_worst*.

7.2 The plot_memory_usage script

This is a Python script which runs Valgrind's massif tool on a few programs, and plots the results. The massif tool records the heap and stack usage of a program.

There are 3 programs, and each of them allocate and deallocate a number of objects with a different allocator. The allocation pattern is the same as in *bench_normal*. Between each allocation and deallocation there is a delay of 1ms. This is because the massif tool takes snapshots every few millisecond. These programs run quite quickly, but if there is a short pause between allocations, then massif can snapshot the memory usage of the program with more accuracy.

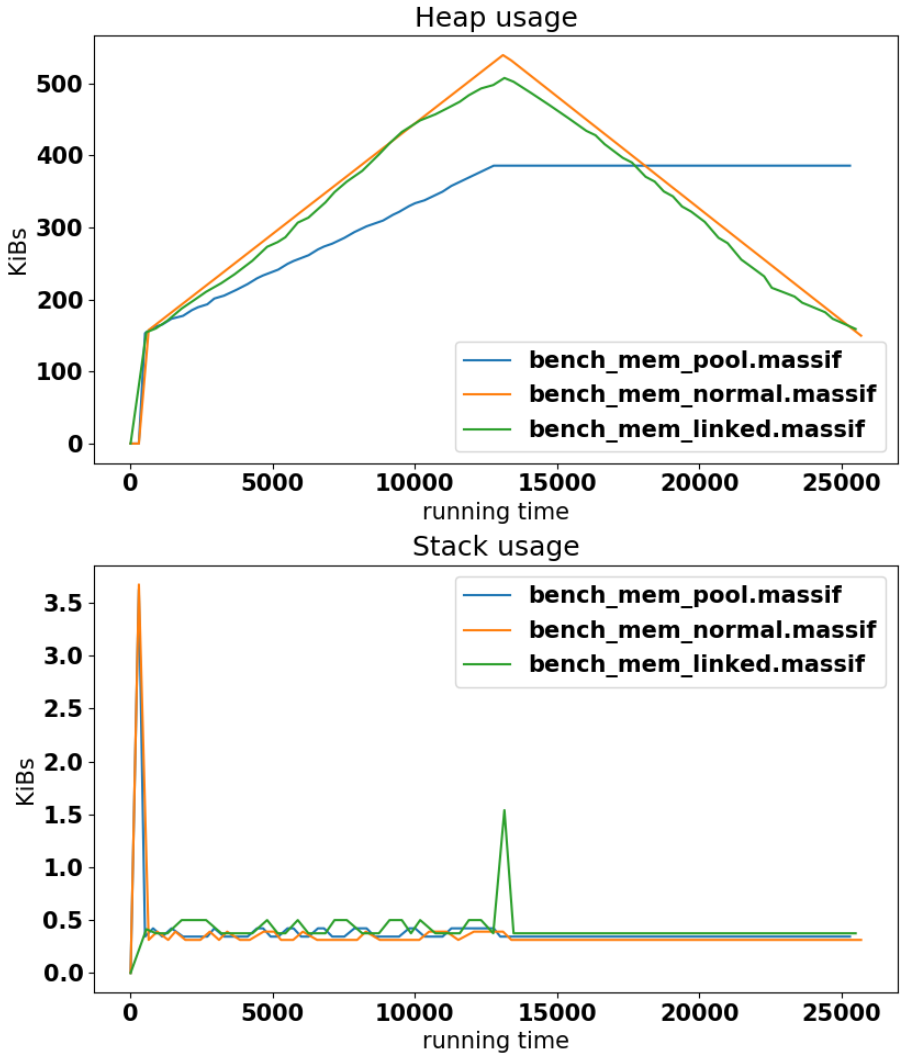

Figure 7.11: Heap and stack usage of the three allocators when allocating 10,000 objects

THe orange line is `new`, the green line is *LinkedPool*, and the blue line is *MemoryPool*.

It can be seen in figure 7.11 that all allocators grow lineraly to their peak. *MemoryPool* uses less memory than the other two allocators. Is uses 394856 KBs of heap memory while *LinkedPool* uses 519440 KBs and `new` uses 551960 KBs. Although the stack usage is very similar, *LinkedPool* uses more stack at the point where the peak is reached, but the difference is negligible.

7.3 The time_alloc_benchmarks script

This script runs a few allocation benchmarks, and plots their memory usage. The benchmarks allocate objects of size 8, 16, 32, and so on until 1024 bytes. Each type of object is allocated as many times as needed until it occupies 0.5 GBs of memory. Therefore the benchmark allocates 4 GBs of memory. There are 9 benchmarks, and they work in the following way: the first benchmark allocates all the objects with `new`, the second benchmark allocates all objects of size 8 with *LinkedPool*, and the rest with `new`, and so on until the last benchmark which allocates all objects using *LinkedPool*.

These benchmarks were used to decide the best threshold value for *Custom-New*.

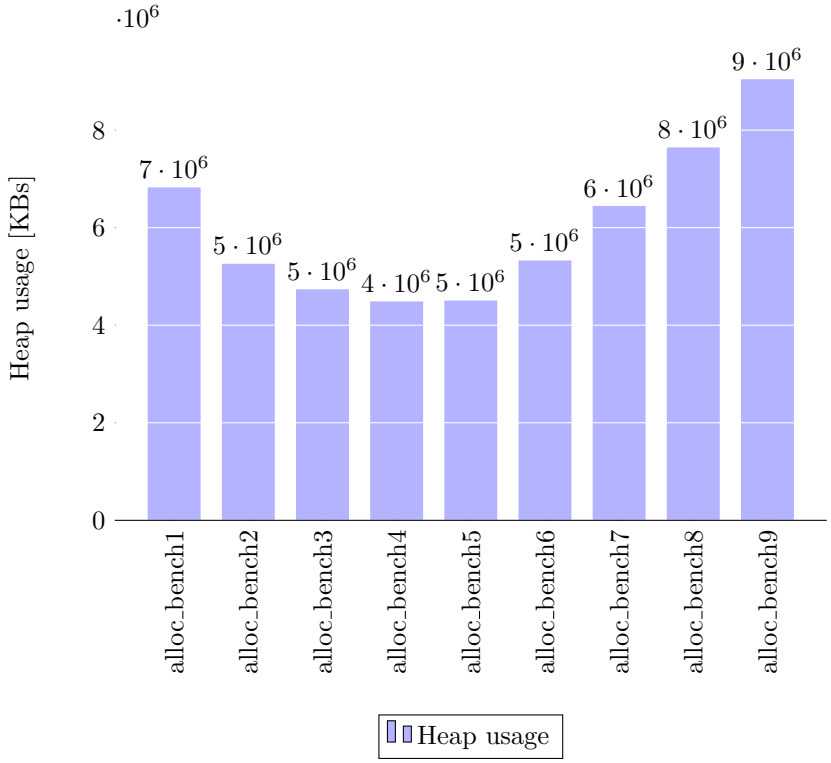

Figure 7.12: Heap memory usage of all allocation benchmarks.

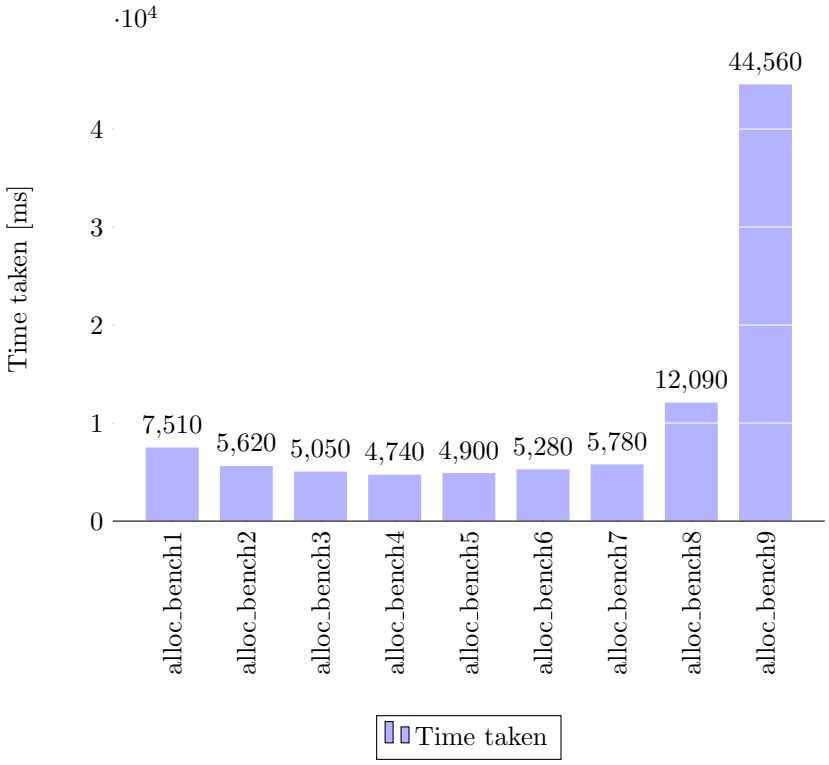

Figure 7.13: The duration of the benchmark.

In figure 7.12 it can be seen that *alloc_bench4* and *alloc_bench5* use the least memory, and in figure 7.13 these two benchmarks execute the fastest. In *alloc_bench5*, all objects of size 8, 16, 32, 64 bytes are allocated with *LinkedPool*. This means that the theshold for `malloc` can be set to 64. However, *alloc_bench6* also performs well, even though it uses slightly more memory. This is why the threshold is set to 128 in *CustomNew*.

Chapter 8

Critical evaluation

8.1 LinkedPool

In chapter 7, it was shown that although *LinkedPool* performs well when compared to `malloc`, *MemoryPool* performs better.

LinkedPool and *MemoryPool* work very similarly. Both allocators use a linked list to keep track of empty slots, but in a different way. The reason why *MemoryPool* is faster than *LinkedPool* is because of its simple implementation. *MemoryPool* works by allocating pages of memory as well, but it does not create a header in each pool. The allocated page is partitioned into slots, and the address of each slot is inserted into a linked list. Each memory pool of *LinkedPool* has its own linked list, but in the case of *MemoryPool*, the implementation keeps track of a global linked list, which contains slots from multiple pages of memory. When the `allocate` method is called, *MemoryPool* returns the first element in the linked list. When a slot is freed, the slot is simply added back into the list. The implementation is simple, and achieves all the goals of a memory pool allocator. However, *MemoryPool* has a significant problem, which *LinkedPool* does not have. In figure 7.11, each allocator's memory usage increases until they reach their peaks. After that, *LinkedPool*'s and `new`'s memory usage starts decreasing to zero, while the usage of *MemoryPool* remains at the value of its peak until the execution ends. This is a problem, because the memory that has supposedly been deallocated is not returned to the system. This means that memory is leaked. This is very bad for programs that rely on releasing the memory of some computation to make space for some other memory-heavy computation. Such programs might run out of memory if they use *MemoryPool*. This is the main reason why that the implementation described in this report uses a pool header. If pools are empty, there is no need to keep them in memory, and can be deallocated.

8.2 CustomNew

It is hard to decide whether the *CustomNew* allocator performs better than `malloc`. Therefore, *CustomNew* is going to be injected into a few programs, and their memory usage will be compared to that of the program run in a normal environment.

8.2.1 alloc_benchmark1

This is one of the benchmarks described in chapter 7. It allocates 4GB of memory, but its total memory usage is a lot higher, as seen in table 7.12. In order to asses the performance of *CustomNew*, this benchmark is first run normally, and then *CustomNew* is injected into it by using the two methods described in chapter 5. The memory usage and execution time of the benchmark is recorded during each execution.

Method	Time taken (seconds)	Memory Usage (KB)
normal execution	4.71	6810616
LD_PRELOAD	4.37	6378032
LLVM	4.71	5319116

Table 8.1: *alloc_bench1* run with the three methods.

Table 8.1 shows that the running time of the three methods is quite similar. The memory usage, on the other hand, is very different. When executed normally, the total memory usage of *alloc_bench1* is 6.8 GBs. After using LLVM to inject the *CustomNew* allocator into the benchmark, the memory usage of the program is reduced to 5.3 GBs, which is a significant improvement. The *LD_PRELOAD* method also performs well, but the issue described in chapter 5 can be seen in this case. Because the alignment is always considered to be 16, the objects will be allocated in different pools than expected. However, the LLVM-based method knows the true alignment, therefore all allocations are placed into the correct pools.

8.2.2 Popf

This is a planner which can be used to find solutions to planning problems. Planners consume a lot of memory when they create and traverse the search space. If the memory usage is reduced, then the planner can consider more states in the search space, which can lead to finding optimal solutions. In order to determine if the pools perform well, the Popf planner is executed normally, and with the two injection mechanism.

Method	Time taken (seconds)	Memory Usage (KB)
normal execution	98.24	2755176
LD_PRELOAD	107.38	4070300
LLVM	104.69	3957336

Table 8.2: Popf run with the three methods.

Table 8.2 shows that the running times are different. The Popf planner executed faster normally than with the injected pools. The memory usage also increased rather than decreased. The reason why this happens is because Popf allocates a large amount of objects of size 17, which are 16 byte aligned. This means that *CustomNew* will round this size to 32 bytes. This makes the program waste 15 bytes per object, which is almost double the amount of the overheads of `malloc` per small object.

Chapter 9

Conclusion

Memory pools are very useful, and can be used to reduce the memory usage of programs. *LinkedPool* and *MemoryPool* provide a way to allocate smaller objects with less overheads than `malloc`. These allocators are also very fast in terms of allocation and deallocation speed, as seen in chapter 7.

Injecting memory pools, on the other hand, usually works in the favour of a program, but it can sometimes make it perform worse, like in the case of Popf. This means that injecting pools might not be such a great idea all the time. The best way to reduce the memory usage of a program is to introduce memory pools for the types that are allocated most often. In order to find out which objects are allocated repeatedly, *customnewdebug* can be injected into a program in order to gather this information. Based on this, the authors can optimize the memory usage by introucing memory pools for the types that are most commonly allocated. This approach might seem tedious, but it provides the best overall performance. The required source code changes are minimal, because typically only a small number of types are allocated very often, which means only a few classes need to be changed.

Bibliography

[1] boost. Memory pool implementation in C++. https://github.com/boostorg/pool/, 2000. Last accessed: 2018-03-22.

[2] Acay Cosku. Memory pool implementation in C++. https://github.com/cacay/MemoryPool, 2013. Last accessed: 2018-03-22.

[3] C++ concepts: Allocator. http://en.cppreference.com/w/cpp/concept/Allocator. Last accessed: 2017-12-10.

[4] EventHelix.com Inc. Byte Alignment and Ordering. https://www.eventhelix.com/RealtimeMantra/ByteAlignmentAndOrdering.htm, 2017. Last accessed: 2018-03-22.

[5] Gloger Wolfram and Lea Doug. Malloc implementation for multiple threads without lock contention. https://code.woboq.org/userspace/glibc/malloc/malloc.c.html, 2001. Last accessed: 2017-12-10.

[6] Lea Doug. A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html, 1996. Last accessed: 2017-12-10.

[7] Lattner Chris and Adve Vikram. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. https://llvm.org/pubs/2005-05-21-PLDI-PoolAlloc.pdf, 2005. Last accessed: 2018-03-22.

[8] Nystrom Bob. Game Programming Patterns: Object Pools. http://gameprogrammingpatterns.com/object-pool.html, 2009. Last accessed: 2017-12-10.

[9] Operator new in C++. http://en.cppreference.com/w/cpp/memory/new/operator_new. Last accessed: 2017-12-10.

[10] Operator delete in C++. http://en.cppreference.com/w/cpp/memory/new/operator_delete. Last accessed: 2017-12-10.

[11] The std::set container. http://en.cppreference.com/w/cpp/container/set, 2017. Last accessed: 2018-03-29.

[12] Belue James. AVL vs. Red-Black: the conclusion. https://nathanbelue.blogspot.ro/2013/01/avl-vs-red-black-conclusion_6.html, 2013. Last accessed: 2018-03-29.

[13] Brito Leonardo. Benchmarking Red-black and AVL trees. https://codedeposit.wordpress.com/2015/10/07/red-black-vs-avl/, 2015. Last accessed: 2018-03-29.