Managing Your Video Game Memory

Albert White

Abstract— In this work, we look at a ready-to-use, durable, and computationally fast fixed-size memory pool manager with no-loops and no-memory overhead that is ideal for time-critical systems like games. This is accomplished by accounting for vacant memory locations and adopting a trouble-free indexing mechanism. We show how it works with simple step-by-step instructions. Furthermore, we compare the memory pool manager's performance to that of a system allocator (e.g., malloc) across a variety of allocations and sizes.

Index Terms— memory pools; real-time; memory allocation; memory manager; memory de-allocation; dynamic memory

1 INTRODUCTION

A high-quality memory management system is essential for any application that makes a significant number of allocations and de-allocations. In hindsight, research has shown that software can spend up to 50

Nonetheless, the majority of apps employ a generic memory management system that strives to provide a best-for-all solution by adapting to every potential case. These tactics are overkill for some systems, such as gaming, where speed is crucial. Instead, a simpler method of partitioning memory into fixed-sized sections known as pools can yield substantial benefits such as higher performance, zero fragmentation, and memory management. As a result, we concentrate on a fixed-pool approach and provide a method for formation and destruction with no overhead and almost no computational cost. It may also be used in conjunction with an existing system to produce a simple hybrid solution. Several instances of various fixed-sized pools, on the other hand, may be utilised to produce

Alternatively, in some time-critical systems, such as games, system allocations are kept to a minimal minimum in order for the operation to perform as quickly as possible. However, for a constantly changing system, memory must be allowed for changing resources such as data assets (graphical pictures, music files, scripts) that are loaded dynamically during runtime. Prior to execution, the sizes of these resources may be known. As a result, the most effective memory pool manager is the fixed memory pool manager. As previously mentioned, a variety of pools may be employed to execute a best-fit strategy for dealing with data of varying sizes.

When a memory pool is formed, it initialises all of its segments. This can be costly since it is normally necessary to loop through all uninitialized segments. Our technique creates with minimal processing cost since we just initialise the first element (i.e., no loops). As a consequence, if a memory pool is only used halfway before being deleted, fewer CPU cycles are wasted. Furthermore, in dynamic memory systems where partitioned memory is continually produced and removed, the cost of setup may be substantial (e.g., pools being repeatedly partitioned into smaller pools at run-time). To summarise, a memory pool may enable an application run quicker, with more control, better flexibility, higher customizability, significantly improved resilience, and decreased fragmentation. To sum up,

The fixed-size pool solution we provide has the following features: No loops (fast access times) There aren't any recursive functions. The initial investment is little. Memory use is really low (few dozen bytes) The algorithm is simple and devoid of errors. No-fragmentation

The remainder of the paper is organised as follows. We begin by reviewing comparable work and continue by outlining this paper's significant distinctions and contribution. Then we'll go over how memory pools work in detail. Then, we concentrate on an unique implementation. We talk about the benefits and drawbacks of our technique. Then, based on a range of tests, we show a set of practical outcomes (to give benchmarking data). We conclude with results and research recommendations.

2 RELATED WORK

Memory management strategies have been intensively researched in recent decades. There are several methods and algorithms available, some of which are quite complicated and difficult to understand. The solution we propose here, on the other hand, is not innovative, but rather a modification of an existing technique in which loops and initialization overheads are eliminated, making the final algorithm incredibly speedy and simple. With a small memory footprint and an O(1) access time, the approach is also one of the most memory efficient ways known. The appendix has a rudimentary C++ implementation.

Memory pools have long been used to accelerate memory allocations and de-allocations in high-performance systems. In order to minimise pre-fetch time, Zhao et al. used memory pools to bundle data from upcoming calls into segregated memory. Applegate's research emphasised the several approaches and benefits of high-performance memory in

portable programmes, as well as the benefits of memory pools. Malakhow goes into further detail on the benefits of memory pools and their use in high-performance multi-threaded systems. While Hanson's single-pool allocator is equal to ours, our technique is more straightforward and easier to adapt for ad hoc implementation. Meyers also discusses performance elements such as macros and monolithic functions, which may be used to increase speed and gain.

This paper makes a contribution by proposing a realistic, basic, fixed-size memory pool manager with no loops, very low memory cost, and computational speed. We next compare the approach to the typical system memory allocator (e.g., malloc) to provide the reader with a real-world computational comparison of the speed differences.

3 METHOD

When compared to a more difficult and generic solution, the comparison shows how much faster a simple and intelligent method can be. We determine what we know and what we need to compute in order to describe how the fixed-size memory pool works (to help make the details more understandable). We get a fixed amount of RAM when we create the pool.

The accounting algorithm keeps track of unused blocks. To locate the utilised blocks, we only need to know which blocks are empty. This list of vacant blocks changes when blocks are allotted and de-allocated. 1. Explain how wasted memory is connected (the unused memory blocks store index information to identify the free space and memory chunking).

However, we save money on startup by connecting all of the idle blocks. Alternatively, we initialise a variable that tells us how many of the n blocks have been added to the unused list. At each allocation, unused blocks are added to the list, and the variable containing the number of initialised blocks is updated.

We can put data in the memory blocks that are being watched since they are not being used. Each unused block preserves the index of the next unused block. The pool keeps track of the head of any unused connected chain. Memory blocks must have a minimum size constraint in order for this accounting system to operate. Each memory block must have a minimum of four bytes. This is due to each unused memory block containing the index of the next unused memory block, resulting in a linked list of all unused blocks. As a result, the index to the next unused block is retained for each unused block, and so on. The head of the first unused block.

During allocation, we simply add new, unused blocks to the list. We keep track of how many blocks have been added to the list and stop adding new blocks when we reach the maximum number. By just initialising blocks as needed, we eliminate loops and starting costs. To summarise, when blocks are assigned, new unused blocks are initiated and added to the list as needed.

A simplified step-by-step representation of the fixed-pool approach in operation. We build a four-brick fixed pool. We demonstrate how unused blocks and member variables change sequentially during the construction, allocation, and de-allocation methods (identifying uninitialized and unknown memory with question marks the three variables used by the pool for bookkeeping). Assurance It might be difficult and error prone to write a bespoke memory pool allocator. While the fixed size memory pool solution is simple to construct, it is recommended that extra verification and sanity checks be included to ensure a robust and trustworthy implementation. These psychological and physiological defences

Benchmarking timings of up to 100 times indicate the various allocation durations of operating within and outside the debugger). The memory pool has the most control and may conduct a variety of custom tests. They may be enabled and deleted at any time, and are less computationally costly than system memory checks, allowing builds to run quickly while collecting debug data.

De-allocated memory addresses, for example, are easily verifiable since each memory address must be inside an upper and lower border of the continuous memory region. Furthermore, the de-allocated memory address must be the same as one of the addresses from the partitioned memory blocks. Memory guards may be extended even further to incorporate boundary checks by including a pre and post byte signature in each block. To discover flaws and offer sanity checks, these memory guards can be tested globally (for all blocks) and locally (for the presently erased block). Leaks can also be discovered by expanding and embedding memory guards to save extra information about the allocation, such as the line number.

4 IMPLEMENTATION

The code was written in C++. The pool was established using create/destroy routines rather than constructor/destructor approaches so that it could be dynamically enlarged without destroying and recreating the pool each time it needed reconfiguration. The implementation includes four main public functions: Create, Destroy, Allocate, and De-allocate. In the appendix, you'll find the source code for implementing the fixed-size memory pool. All validation and sanity check code has been removed to keep the source code as simple and comprehensible as feasible.

5 INITIALIZE POOL

A memory block is assigned or obtained: 1. Save the start address, number of blocks, and number of uninitialized unused block Allocator. 2. Look for any open blocks. 3. If required, start the list and add any unused memory blocks. Return to the top of the list of unused blocks. 5. Make the block number from the list's unused block's head the new head. 6. Find the address of the former block head. 7. Verify the memory address once more. 8. Find out what the memory address index id is. 9. Rename it to the index id of the currently unused block's head and make it the head.

Combining the fixed pool allocator with an existing memory management system in C++ by overloading the new and

delete operators would result in improved performance with the least amount of interruption, because dynamic memory management can occupy 38

Furthermore, considerable care must be taken to ensure that classes and structures in C++ that are allocated and de-allocated by the fixed-size pool allocator have their constructors and destructors explicitly called.

6 LIMITATIONS

The fixed-pool memory manager requires a continuous block of memory. If the allocated memory block is dispersed, this can be a major limiting issue. Furthermore, we concentrated on the technique rather than the hardware constraints. For example, a page defect might cause access times to be 10,000 times slower than usual. Furthermore, no mention of leveraging the memory pool in a multi-threaded context has been made. This raises concerns about scalability and how the memory manager can be managed across several cores. Furthermore, the memory pool solution given is confined to direct memory access systems and so cannot be used in managed memory setups (e.g., Java).

The fixed-size pool allocator's memory requirements may cause two primary issues. For starters, if the requested memory is significantly lower than the slot size, a significant amount of memory will be squandered. Second, and most importantly, memory from the pool cannot be allocated if the requested memory exceeds the slot size. An ad hoc approach may be utilised to address these issues while reducing memory waste and misallocations. In this case, a general system allocator combined with a large number of fixed-size pools may assist in reducing memory waste while still benefiting from pool speedups. However, it should be noted that a big memory management system may become slower and fragmented over time.

7 RESIZING

The fixed-size memory pool keeps a list of unused memory blocks. This list is kept in unused memory and is gradually extended each time a memory block is allocated. As a result, if more memory blocks are required than are available, and more memory is required after the continuous memory pool allocation is complete, the pool may be swiftly and inexpensively extended by altering its member variables. Following the establishment of the member variables, the new memory space will be automatically enlarged and filled during block allocations. At the present, the algorithm always allocates the next unused memory block. An extra check, however, can be added to prohibit the activation of any further redundant blocks.

8 EXPERIMENTAL RESULTS

The technique itself is simple, with no loops, recursion, or processing overhead, and it yields extremely rapid allocations and de-allocations. We created and de-allocated a variety of memory chunks to test how much quicker the memory pool manager is compared to a generic memory system. According to statistics, the fixed-pool allocator is ten times quicker than the traditional system allocator and a thousand times faster when run in debug mode.

9 CONCLUSION AND FURTHER WORK

We demonstrated a simple, unsophisticated, raw-and-ready memory pool solution that provides extremely fast speeds with no overhead and has the added benefit of being simple to comprehend and perform. The fixed-size memory pool is the ideal solution for programmes such as games, which presume that memory allocations occur seldom and that when they do, they are of deterministic size and must be exceedingly quick (for example, graphical assets, particles, network packets and so on). Because the suggested technique is a fundamental building block for developing a more complex and flexible memory manager, an elegant and straightforward approach employing a fixed-size memory pool provides a more robust and scalable solution. However, extra research might be conducted to investigate additional data for the algorithm.

REFERENCES

[1] Ben Kenwright. Fast efficient fixed-size memory pool: No loops and no overhead. *Proc. Computation Tools. IARIA, Nice, France*, 2012.