High Performance External Heap Allocator FDD

v1.0

# High Level

The external memory allocator ‘manages’ external memory with book keeping data that is stored in main memory. Our allocator for **large** allocations is a best fit allocator and for **small** allocations it is a fixed size pool allocator. Every small bin manages 1 specific size and uses a piece of memory the size of a **page**.

Main Memory – Book keeping data

Configuration

Small Bin

Large Bin

Allocation larger than Y

Allocator

A best fit allocator with small bins for small allocations. The large bin allocator is a best fit allocator using a BST (red-black tree) for storing the free blocks. The small bin allocator is a fixed size pool allocator. Every small bin is using one **page** (by default is 65KB). The large bin manages multiple continues pages. The small bins are allocated from the beginning or the end of the managed memory and they should never fragment the large allocator.

External Memory

This can be Video, Sound or even plain Main memory.

Small Bin

Small Bin

Small Bin

# Small Bin

The default small bin allocator will create 6 small bins that handle sizes 32, 64, 128, 256, 512 and 1024. The user is able to specify a number of small bins and the range doesn’t have to be continues, this example is possible: {64, 256 and 1024}.

## 2.1 Functionality

Small bins are handling allocation requests of small sizes where every size will be aligned up to their next power of 2. For example we could have 6 small bins managing small allocation requests of the following sizes; 32, 64, 128, 256, 512, 1024. Every allocation larger than 1024 will go to the large bin and any allocation smaller than 32 will go to the smallest small bin, in this case ‘32’.

Small bins are managed using a BST for free blocks with the size as the key and a BST for free blocks with the address as the key. Since we are managing external memory our book keeping data actually holds the pointer to the place in the external memory. We do this so that when we have to de-allocate we can **find** the appropriate book keeping data.

## 2.1 Limitations

When a small bin becomes full a new page needs to be allocated. One constraint here is that we don’t want to fragment the large allocator by taking a page somewhere in the large bin, we prefer to take a page from the beginning or end of the large bin memory, doing like this means that we avoid fragmenting the large bin. When we fail to allocate a page from the large bin one fallback strategy could be to allocate from the next small bin.

# Large Bin

The small allocator is simple and fast, however, as allocation sizes grow, the benefits of binning and pool allocations quickly disappear. To combat this we switch to a different allocator that uses a red-black tree to manage the free nodes. A red-black tree has several nice properties that would be helpful in our scenario. First, itself balances and thus provides a guaranteed O(log(N)) searches where N is the number of nodes. Second, it also provides a sorted traversal which is very important when dealing with alignment constraints.

## 3.1 BST

Our BST is a red-black tree and has a straightforward implementation, as in [Cormen90], with few notable modifications.

Parent (BST)

Node

Node

Node

Node

Node

Right (BST)

Left (BST)

As shown in Figure 4, we have the classic left, right and parent pointers. We also have a sibling pointer that forms a linked list of blocks that have the same size. This helps tremendously with performance, since it’s quite common to have lots of free blocks with identical sizes. In contrast, a traditional red-black tree would store same key value nodes as either left or right children, depending on convention. This would predictably reduce performance since when searching through these nodes, the search space is not halved as usual to achieve O(log(N)) speed but is merely walked in a linear fashion of O(N).

The left/right childe pointers are organized as an array of two entries. This is done mostly to simplify operations such as “rotate left”, which normally have mirrored counterparts such as “rotate right”. Using an index to signify left or right we can then have a generic version that can become either.

Furthermore, in each node we keep information on which side of its parent that node is attached – left or right, as well as its “color” – red or black.

The parent side index is quite important for performance, especially when combined with a red-black tree that uses the so-called “nil” node, since the essential “rotate” operation can then become completely branch free.

Nil Node

Root Node

Node

Node

Node

Node

Node

As shown in Figure 5, the “nil” node is a special node that all terminal nodes point to, and is also the node to which the root of the tree is attached. The fact that the root is to the left side of the “nil” node might appear random, but in fact this is very important for traversal operations. It is easy to notice that running a predecessor operation on the “nil” node would give the maximum element in the tree, which is exactly what we want to happen when we iterate the tree backwards starting from the “nil” node - in STL terminology that would be the “end” of the container. This way the predecessor operation doesn’t need to handle any special cases.

With all this setup done, during an allocation we search the red-black tree for the appropriate size. If the acquired block is too big, it is chopped up and the remainder is returned to the tree. If there are no available blocks this results in an allocation error. To the user we return a pointer to the ll\_node with the lowest 2 bits set so that we can identify that is a handle plus the user cannot accidentally use it because it is unaligned. During a free operation, we cast it to a ll\_node and use the block pointer to get the bin pointer, with the lowest bit we know if it is a small or large allocator block, if it is a large allocation block we add it to the ‘free pointer tree’ and determine if any of the neighbors are prev/next neighbors so they can be coalesced. Since the coalescing is done for every freed block, there could be no more than one adjacent free block on each side, and thus this operation needs to check only two neighbors. The resulting free block is then added to the ‘free size tree’.

The more interesting use of the BST happens when we need to allocate with non-default alignment. We use the fact that we can iterate the BST in sorted order, and we notice that we need to check nodes with sizes equal or larger than the requested size, but smaller than the requested size plus the requested alignment. We then use the binary tree operations “lower bound” with the requested size and “upper bound” with the requested size plus alignment. We then iterate through this range until we find a block that satisfies the alignment constraint. Iterating through a BST is a O(log(N)) operation, so obviously larger alignments would take longer to find.

The important thing to notice is that this will guarantee the smallest-fit block criteria, which is considered to be one of the major factors in reducing fragmentation, something that the traditional approach of over-allocating and shifting doesn’t satisfy very well.

# PS3 Graphic Memory Statistics

Tracking the graphic memory allocations on PS3 shows that the following alignments are used:

* 16
* 64
* 128
* 65536

## C++ Design Code

/// High Performance External Heap Allocator

/// ----------------------------------------

/// This allocator manages external memory like video or sound memory.

/// All bookkeeping data is in main memory so there is 'extra' memory

/// required for this allocator.

/// We can pre-allocate all necessary bookkeeping data from initialization

/// parameters given by the user.

/// You can configure the allocator with:

/// - deallocate in 1 or 2 stages;

///   1 stage  : deallocate will treat the block as free for allocation.

///   2 stages : first deallocate call will do some logic to free the block

///              but it will still be locked from using it for allocation.

///              second deallocate will unlock it for allocation.

/// - the page size

/// - the number and sizes of small bins

/// - the default alignment of the large bin

/// - the size alignment of large allocations

///

/// This allocator manages many small bins and one large bin. The number

/// of small bins can be configured as well as the allocation size that

/// every bin should manage (e.g. 64/128/256/512/1024).

/// All the larger allocations go to the large bin. You can specify the

/// minimum alignment for the large bin as well as the page size.

/// The user also needs to supply a function for copying external memory to

/// facilitate reallocation.

/// Memory block info struct

struct block

{

void\* mPtr; /// Start of this free/used memory

u32 mSize; /// Size of free/used memory from @mMemoryPtr

void\* mBin; /// Either a smallbin\* or largebin\* (bit 0, 0=smallbin, 1=largebin)

block\* mList; /// In a tree of free nodes this will link nodes of the same size

};

/// Linked list node (16 bytes)

struct ll\_node

{

u32 mLock; /// Lock-Locked(1)/Unlocked(0)

block\* mBlock;

ll\_node\* mNext;

ll\_node\* mPrev;

};

/// A handle is actually a pointer to a ll\_node with the 2 lowest bits set to 1

inline bool gIsExternalMemHandle(void \*p) { return ((u32)p&3) == 3; }

inline void\* gHandleToExternalMemPtr(void\* p) { return ((ll\_node\*)((u32)p&~3))->mBlock->mPtr; }

/// BST node (16 bytes)

struct rb\_node

{

rb\_node\* mParent; /// (rb\_node) (bit 0 = Side-Left/Right)

rb\_node\* mChild[2]; /// (rb\_node) Tree children, Left (bit 0 = Color-Red/Black) and Right

block\* mSibling; /// (block) Minimum of 1 sibling

};

/// Small bin (16 bytes)

struct smallbin

{

rb\_node\* mFreeTreeByAddress; /// (rb\_node) BST organized by address to support coalescing during deallocation

ll\_node\* mFreeList; /// (ll\_node) List of free nodes

smallbin\* mNext; /// (smallbin) List of bins

smallbin\* mPrev;

};

// Large bin (8 bytes)

struct largebin

{

rb\_node\* mFreeTreeByAddress; /// (rb\_node) BST organized by address to support coalescing during deallocation

rb\_node\* mFreeTreeBySize; /// (rb\_node) BST organized by free size to support allocation

};

// Small bin allocator

struct smallbin\_allocator

{

u32 mNumSmallBins; /// Number of small bins

u32\* mSmallBinSizes; /// Size of every small bin

smallbin\*\* mSmallBins; /// Small bins

smallbin\*\* mFullBins; /// Full bins

};

/// Allocates from high to low

struct largebin\_allocator

{

u16 mDefaultAlignmentAddress;

u16 mDefaultAlignmentSize;

largebin mLargeBin;

};

struct hph\_ext\_allocator

{

x\_iallocator\* mAllocator; /// Where we and our resources are allocated from

x\_iallocator\* mNodeAllocator; /// Fixed size pool allocator for rb\_node/ll\_node/block/largebin/smallbin/smallbin\_allocator/largebin\_allocator, element size = 16 bytes

/// Configuration

bool mTwoStageDeallocation;

u32 mPageSize;

/// Small bins

smallbin\_allocator mSmallBinAllocator;

/// Large bin

largebin\_allocator mLargeBinAllocator;

/// Doubly linked list of allocations

ll\_node\* mAllocations;

ll\_node\* mDeallocations; /// 1st stage deallocation

/// e.g.

/// u16 sizes[] = { 64, 128, 256, 512, 1024 };

/// init(sizeof(sizes)/sizeof(u16), sizes, 4000);

/// 4000 allocations should include the overhead of holding on to deallocated blocks

/// that cannot be merged. Main memory for 4000 allocs will use 4000 \* 2\*16 = 128.000 bytes.

/// Any free memory chunk will use 3 \* 16 = 48 bytes

void init(u16 num\_small\_bins, u16\* small\_bin\_sizes, u16 max\_num\_allocs);

/// Allocate returns a handle, get the pointer to external

/// memory as follows:

///       void\* handle = externalMemoryAllocator->allocate(256, 16);

///       void\* external\_memory\_ptr = gHandleToExternalMemPtr(handle);

///       externalMemoryAllocator->deallocate(handle);

///       externalMemoryAllocator->deallocate(handle); /// If it is a 2 stage deallocation

void\* allocate(u32 size, u32 alignment); /// Returns a handle

void deallocate(void\* p);

};