Game Developers Conference®

February 28 - March 4, 2011
Moscone Center, San Francisco

www.GDConf.com

UBM
TechWeb

GDC 25

Multi-Core Memory Management Technology in Mortal Kombat

Adisak Pochanayon
Principal Software Engineer
Netherrealm Studios
adisak@wbgames.com

http://twitter.com/adisak
http://facebook.com/adisak

The MK Memory Manager

Q: What is the MK Memory Manager ?

A: Completely new Modern Memory Manager developed with console ideology in mind.

Spring 2011 Mortal Kombat

Topics to cover

- Memory Managers in our previous Game
- Locking and Fixed-Backstore Issues
- Multicore Awareness
- General Architecture and Primary Hybrid Heap
- Small Block Memory Manager
- Simple Lockfree Primitives and Allocators
- Debug Support and Early-Init
- Postmortem Summary

The starting point

"MK vs DC" primarily used two memory managers:

- Unreal Memory Manager (FMalloc)
  - Engine side resources
  - C / C++ memory management
- "Game" Memory Manager
  - Game side resources
  - Console oriented

Unreal MemMgr Limitations

- LibC++ feature set
- No multiple heap support
- Not natively threadsafe / multicore
  - Non-threadsafe memory allocators are protected with a “global lock”
  - “MK vs DC” used DLMalloc internally
- Some operations cause large stalls!

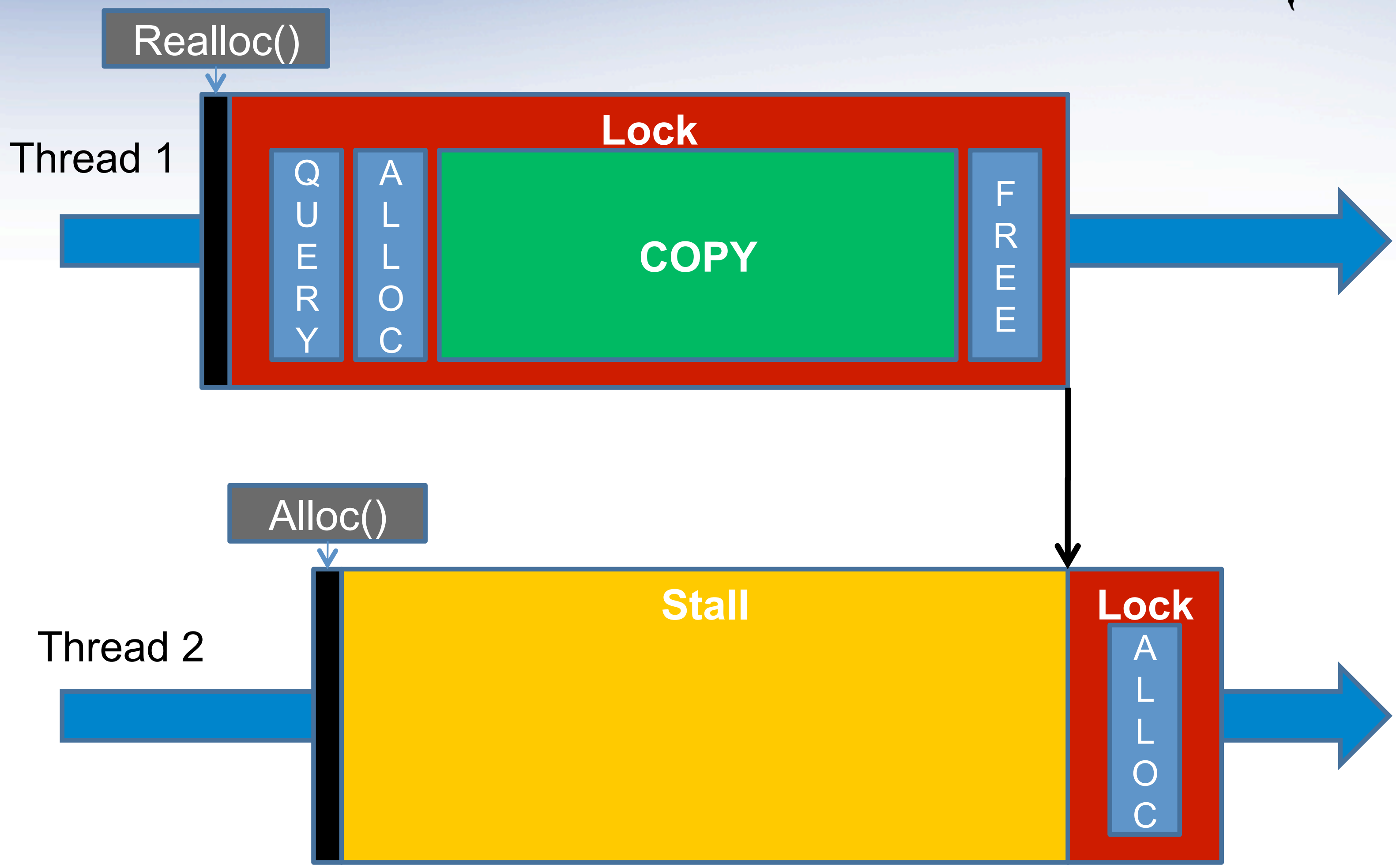
Game MemMgr Limitations

- Not thread safe
- Not “Virtual Memory Aware”
  - Supported only static fixed backstore
- Very Slow / O(N) ops
- Fragmented Easily (naïve first fit)

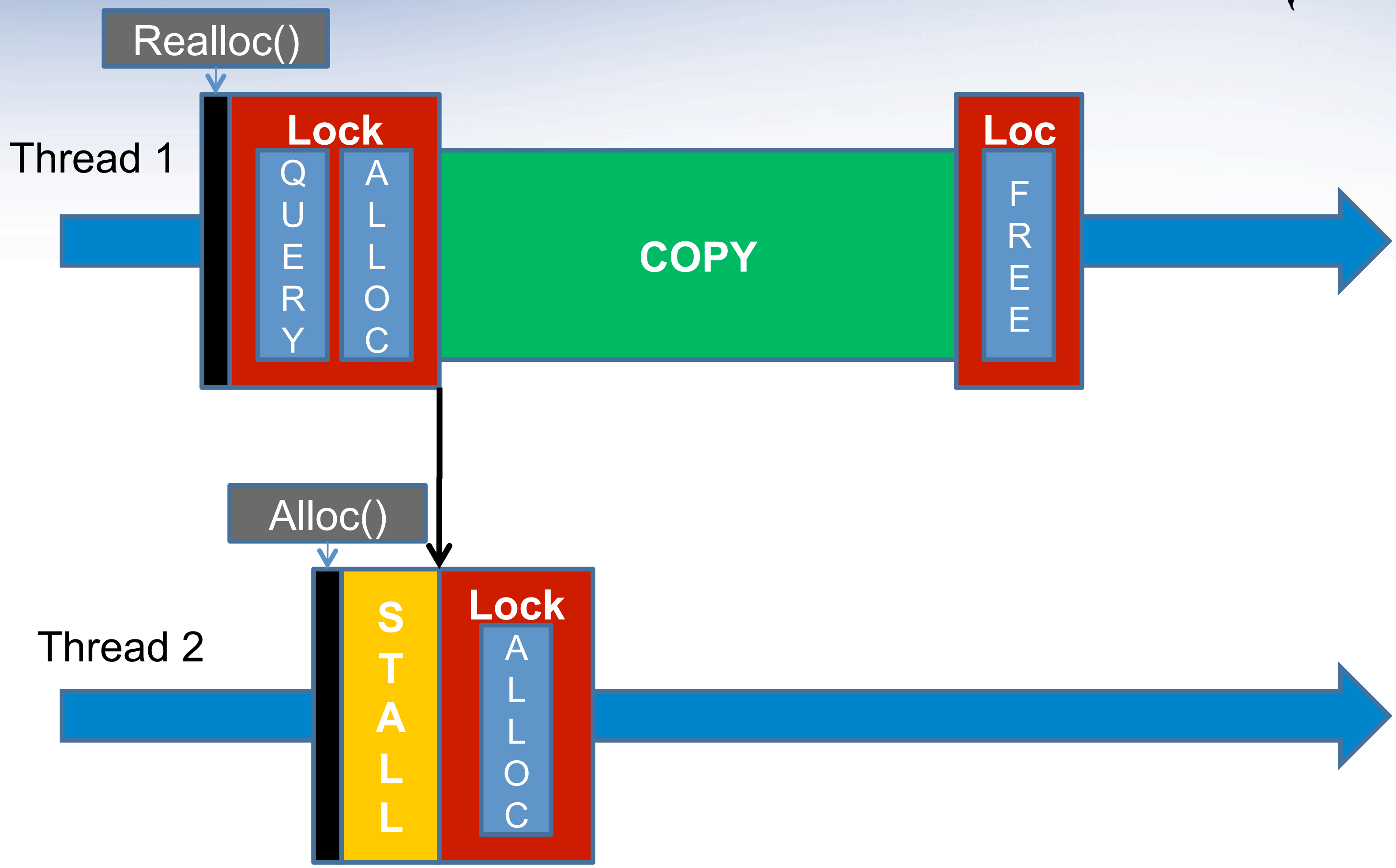
Global Locking is Bad

- Not multicore optimized
- All operations can cause minor stalls or context switches on other threads
- Certain operations can cause large system-wide stalls
  - Large Application Alloc Requests
  - Heap Backstore Allocations
  - Realloc() operations

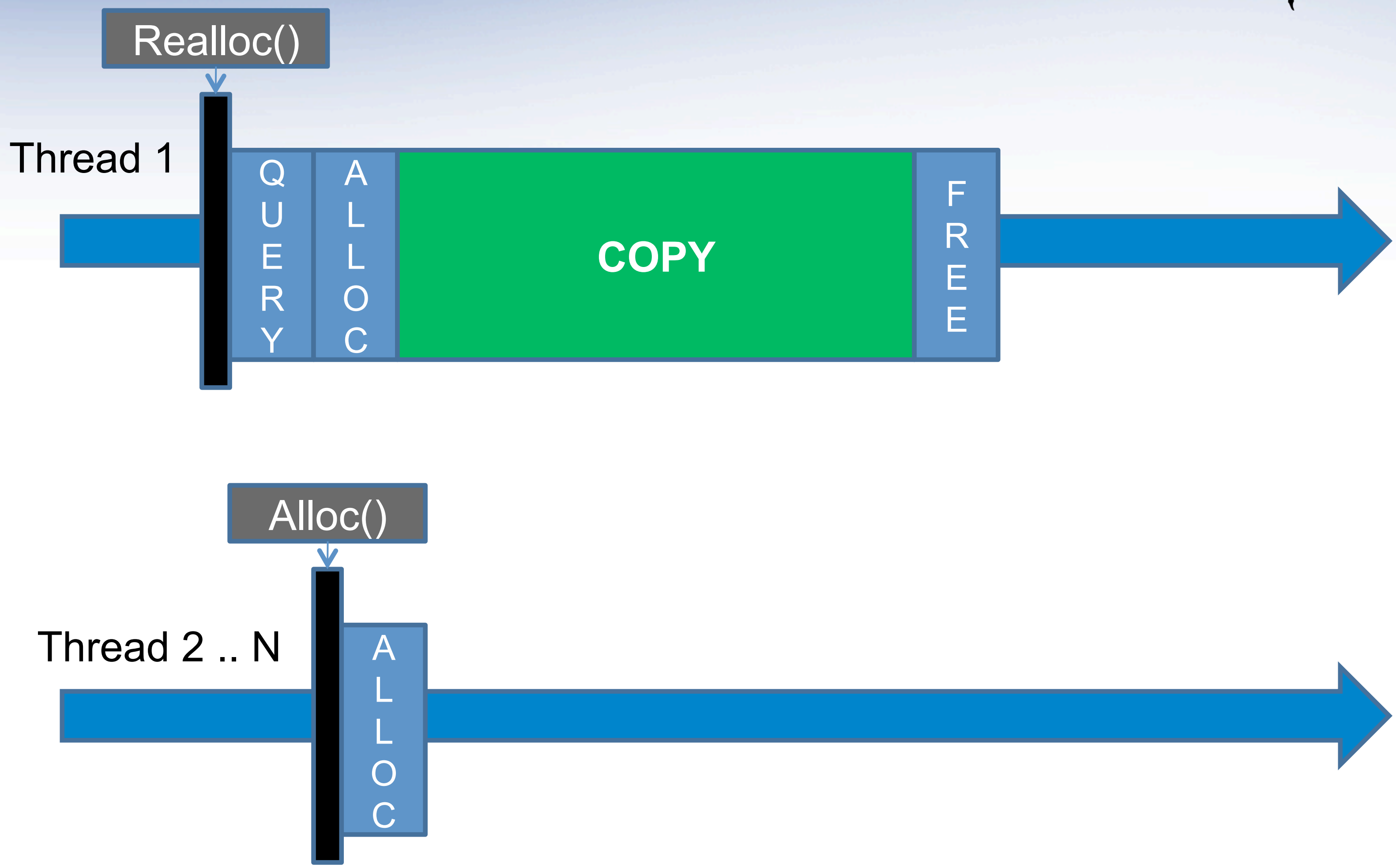
Global Lock Realloc()



Fine-Grained Lock Realloc()

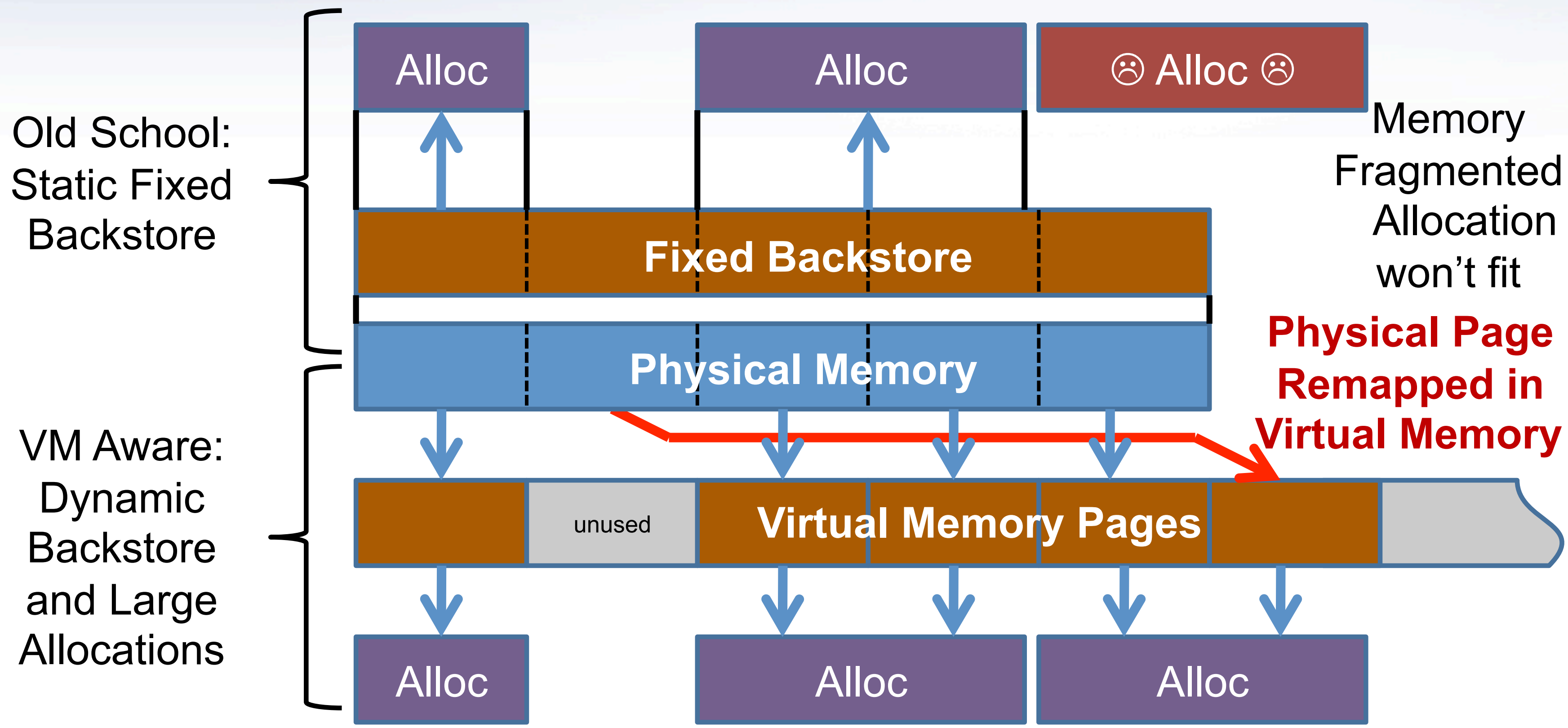


Non-Blocking Realloc()



VM Awareness

Fixed Backstore Leads to Fragments



Virtual Memory "solves" Physical Fragmentation

Multicore Approach

- Threadsafe by default
- Lock-free when possible (and straightforward)
- Prefer Non-blocking locks when required
  - Non-Exclusive Locks (ex: Reader-Writer)
  - Fine-Grained Locking
  - Striped Locking
- High performance for single-threading as well
  - Uncontested accesses do not pay a significant penalty.

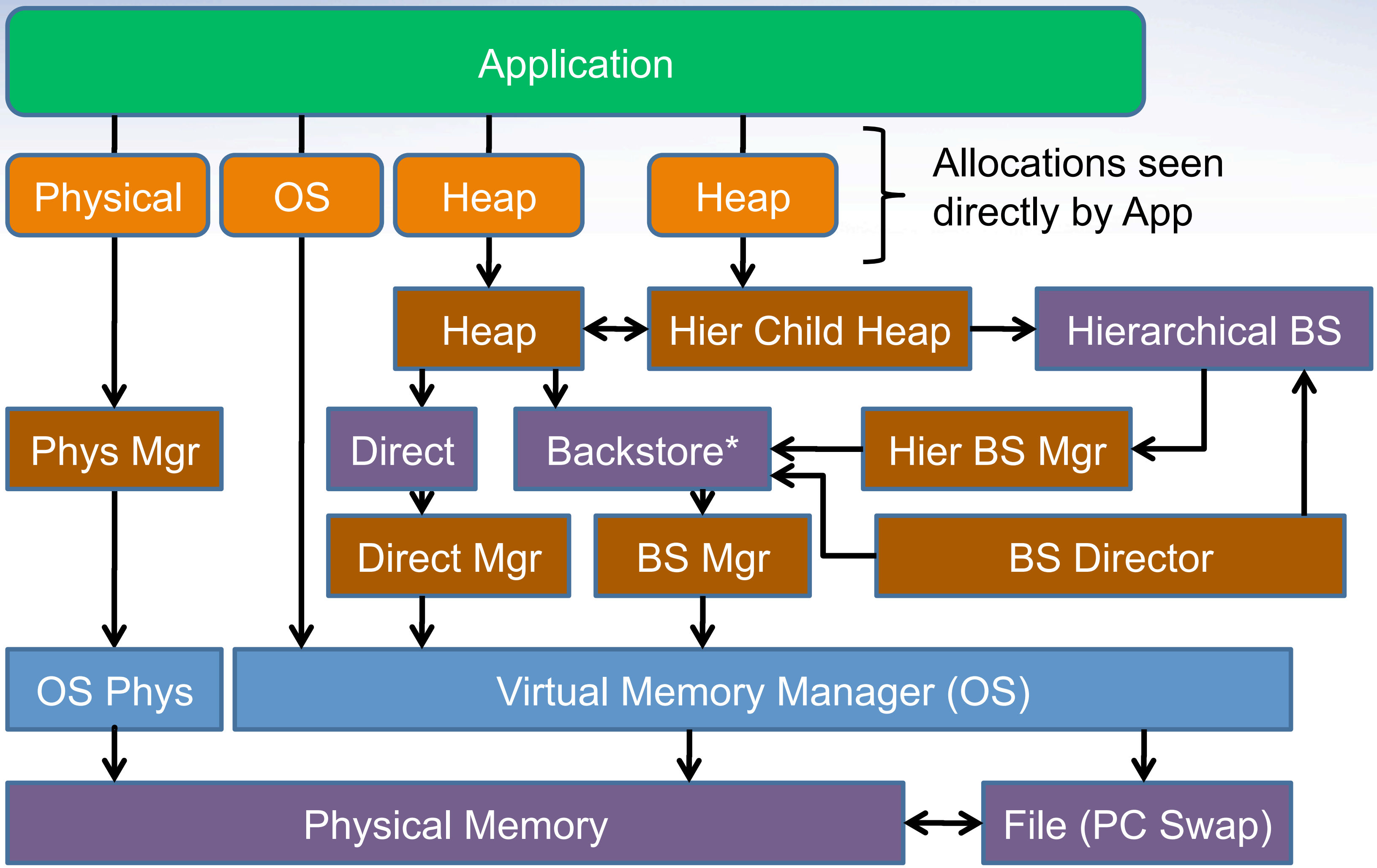
New Memory Manager

- Make Thread-Safe and Multi-Core Optimized
- Unify Separate MemMgr’s for Game and Unreal Engine
- Support multiple heaps with extra features
- Improve performance (both CPU cycles and Memory Usage Efficiency)
- Common Tracking and Debugging Utilities

Concurrent Heaps

- Heaps have minimal Thread “crosstalk”
- Simultaneous allocations / frees from multiple threads possible on a single heap (if supported by heap type – most do!)
- Backstore and Internal Heap Querying operations typically operate concurrently (using Lock-free, Striping or Reader-Writer Locks)
- Realloc ( )’s NEVER block while copy occurs

Simplified Architecture



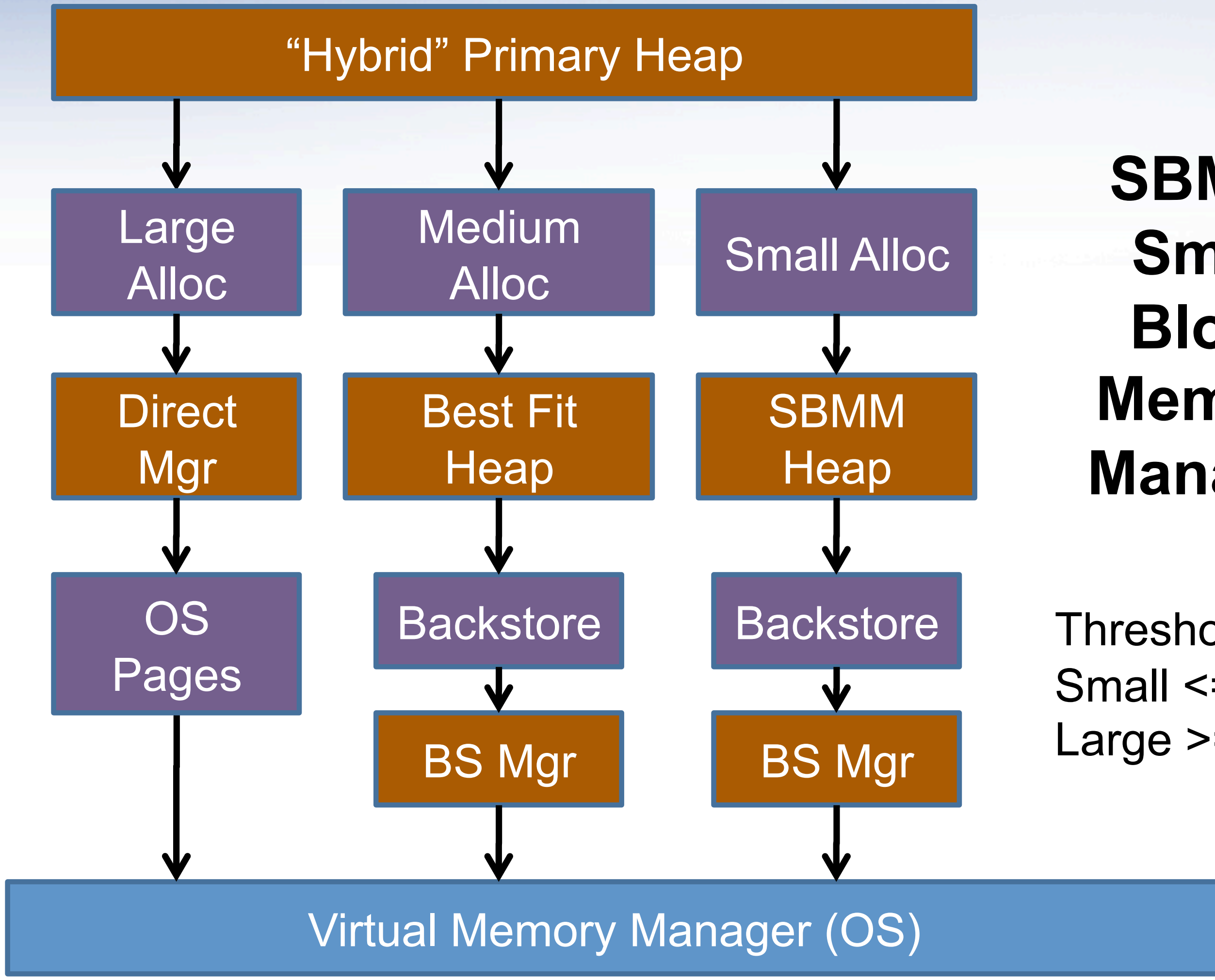
Heap Implementations

- Heap API uses virtual functions
  - Common support API for Backstore and OS Allocs
    - Global Free( ) “knows” to which heap memory is returned
- Easy to make different Heap Implementations
  - Direct OS Heap
  - Best Fit Heap (using Red-Black Tree)
  - Small Block Heap (Lock-Free Alloc / Striped Free)
  - Fixed Block Heap (Lock-Free – used for MK Game Objects)

Hybrid Primary Heap

- Primary Heap uses Hybrid approach to handling allocations
  - Large Allocations go directly through OS to minimize fragmentation (but are tracked internally)
  - Medium Allocations go to a Best-Fit heap
  - Small Block Allocations are handled by their own heap
- C++ new / delete & C malloc / free calls routed to the Primary (Hybrid) Heap.

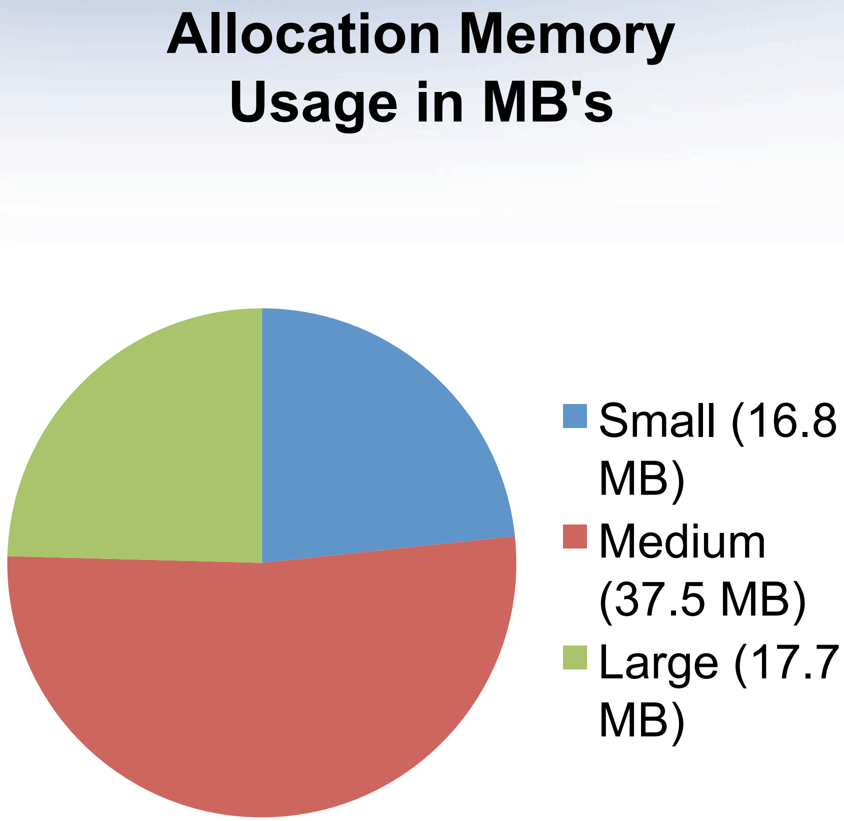
Hybrid Primary Heap

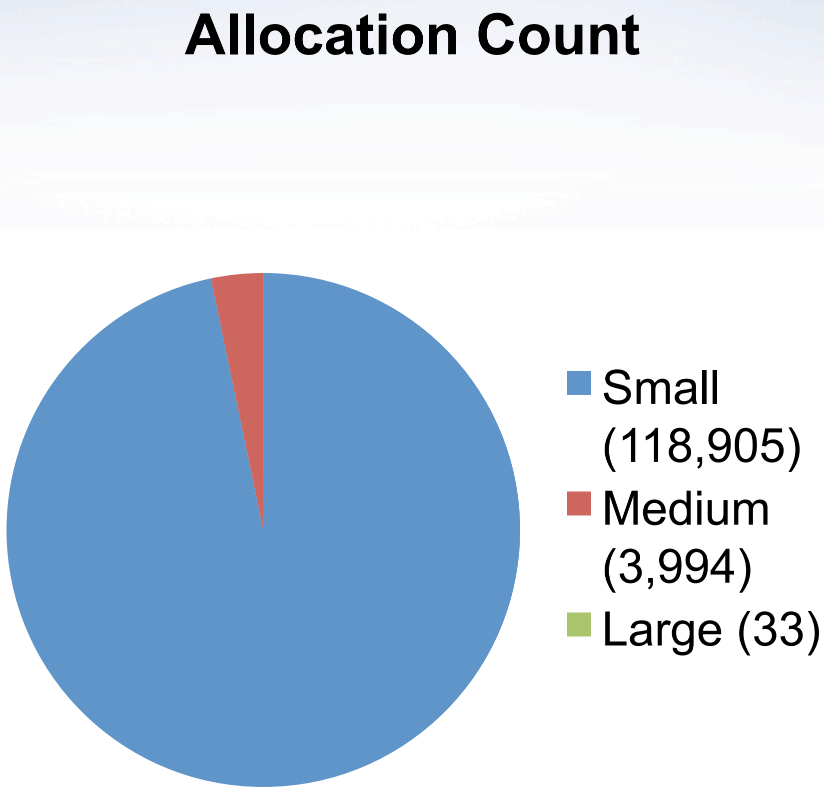


SBMM: Small Block Memory Manager

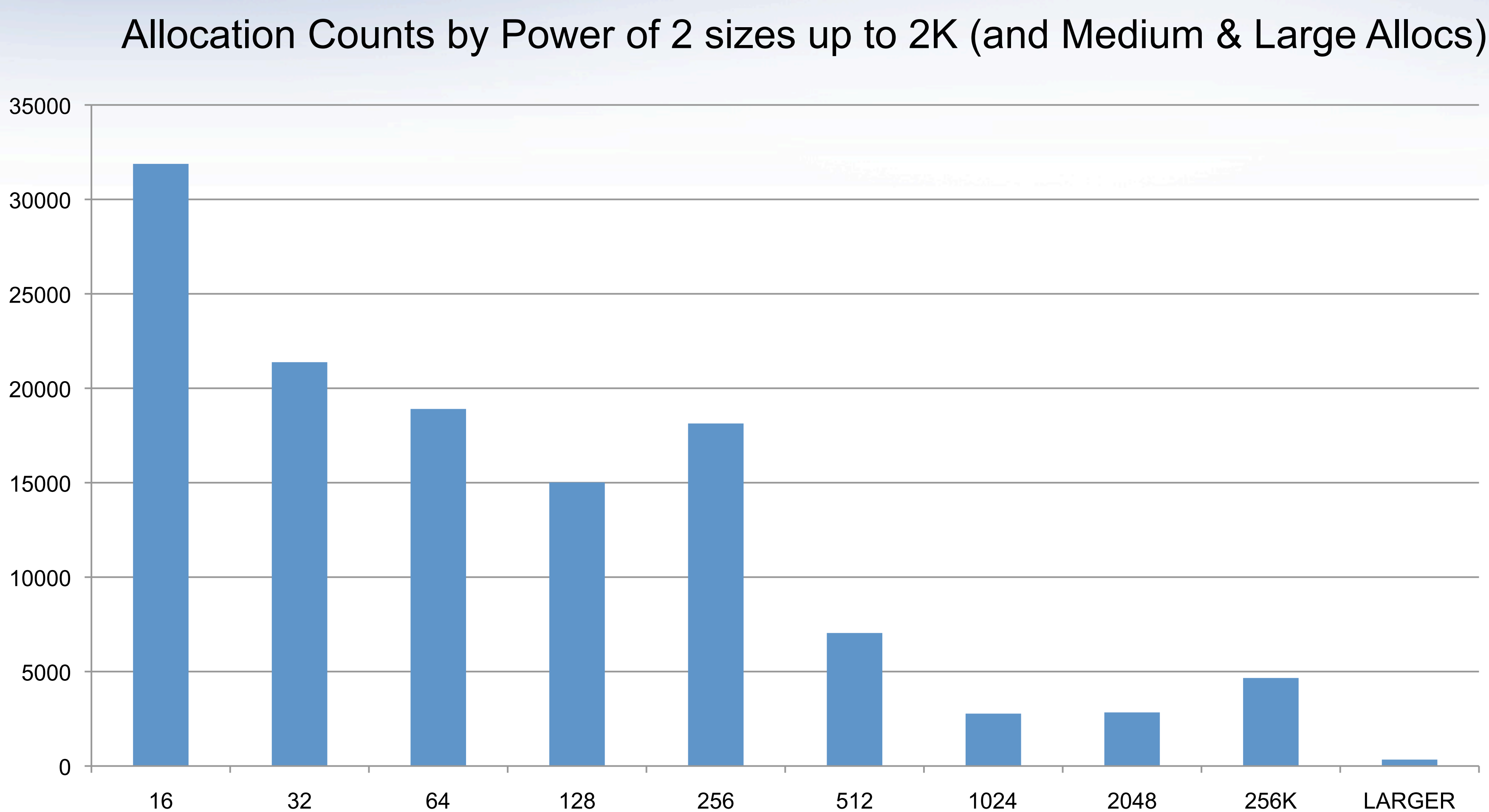
Thresholds:
Small <= 2K
Large >= 256K

Allocation Profiling





Allocation Profiling



Small Block Memory Manager

- SBMM = Small Block Memory Manager
  - Very low thread contention
  - Supports many simultaneous operations
  - Binning allocator
    - Sized Bins
    - Lock Striping = Lock Per Bin
  - LockFree Alloc( )* (*most of the time)
    - Lookaside cache uses "victim" blocks for lockfree Allocs( )
  - Fast Stripe-Locked Free( )

Small Block Memory Manager

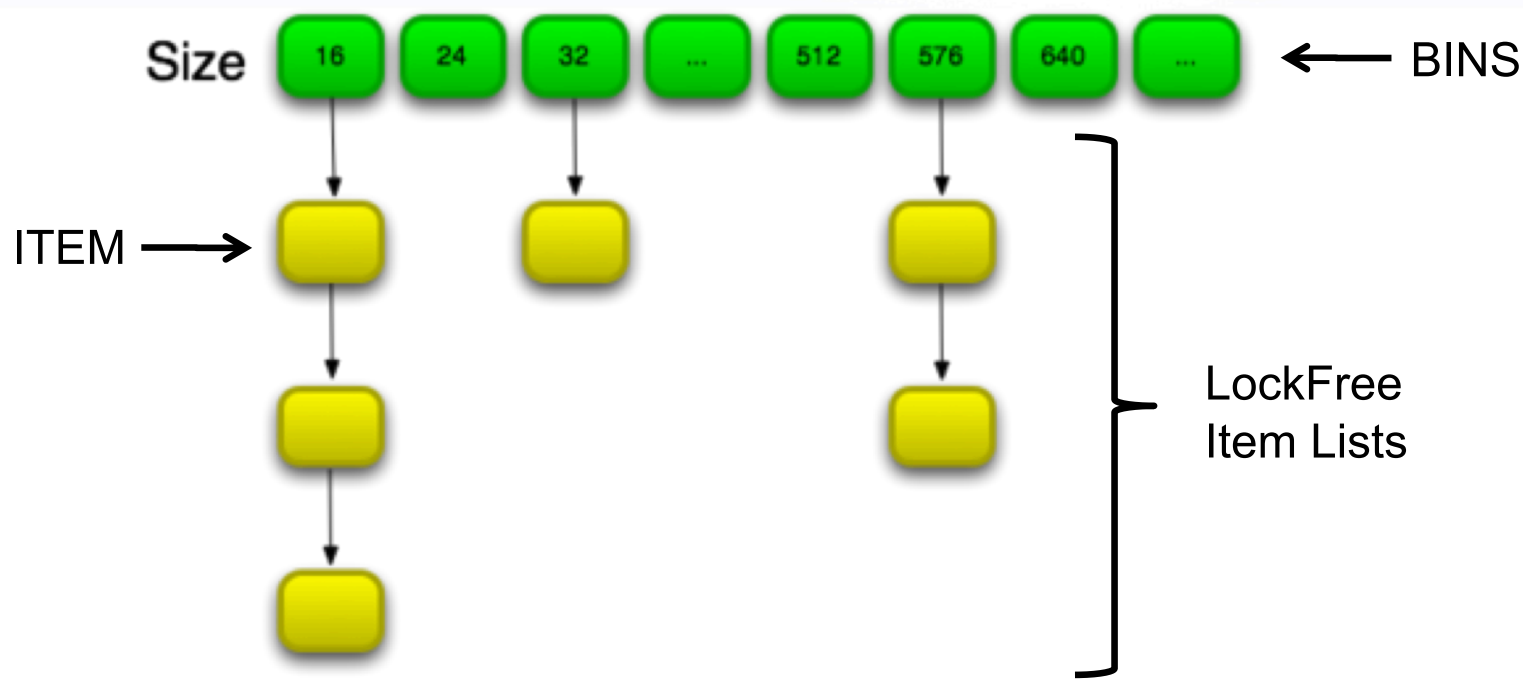
Quick Terminology

- **Bin** = Everything related to Allocations of a Specific Size
- **SuperBlock** = Backstore Memory Chunk (from OS)
- **Block** = Subdivision of SuperBlock. Either empty or owned by a Bin (and containing many items, all of the same size).
- **Item** = Subdivision of Block (sized for a bin). Items represent the actual memory returned from SBMM.
- **Victim** = Lockfree Lookaside cache for a Block's Items

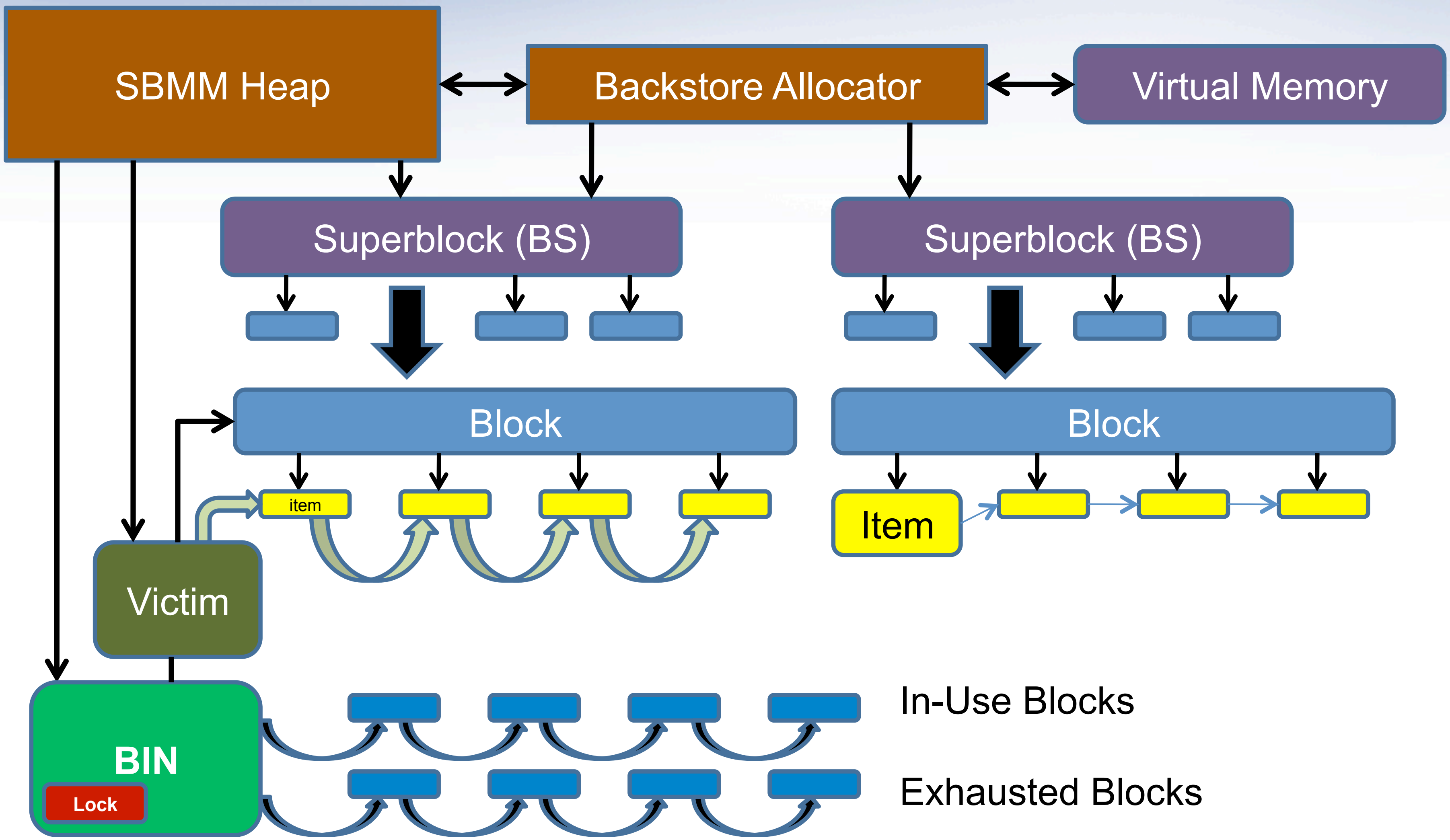
SBMM Binning

"Victims" Look-aside Cache for Allocation

Array of LockFree Lists of Items



SBMM Memory Layout



Small Block Memory Manager

- Mostly LockFree Alloc ( )
  - LockFree freelist cache of “Victim” Block’s Items
  - When empty, Bin striped-lock is acquired and new freelist is established from next Block with free Items
  - This is a very fast operation until all the Blocks are exhausted.
    - In this rare case, a new Block must be taken from the SuperBlocks and a freelist initialized for the items. If all the SuperBlocks are exhausted, a new SuperBlock is requested from OS.

Small Block Memory Manager

- Free( )
  - Originally LockFree but required Delayed GC
  - Striped Lock == Easy Trimming (No Delayed GC)
    - Find Block & Bin Size and Fast-Lock Bin
    - Push memory item and check count
    - *If Trimming required, pull Block, Release Lock, Trim
    - Otherwise Release Lock
    - Uncontested case is very similar to LockFree speed
    - Striped so normally Uncontested

Simple LockFree Allocators

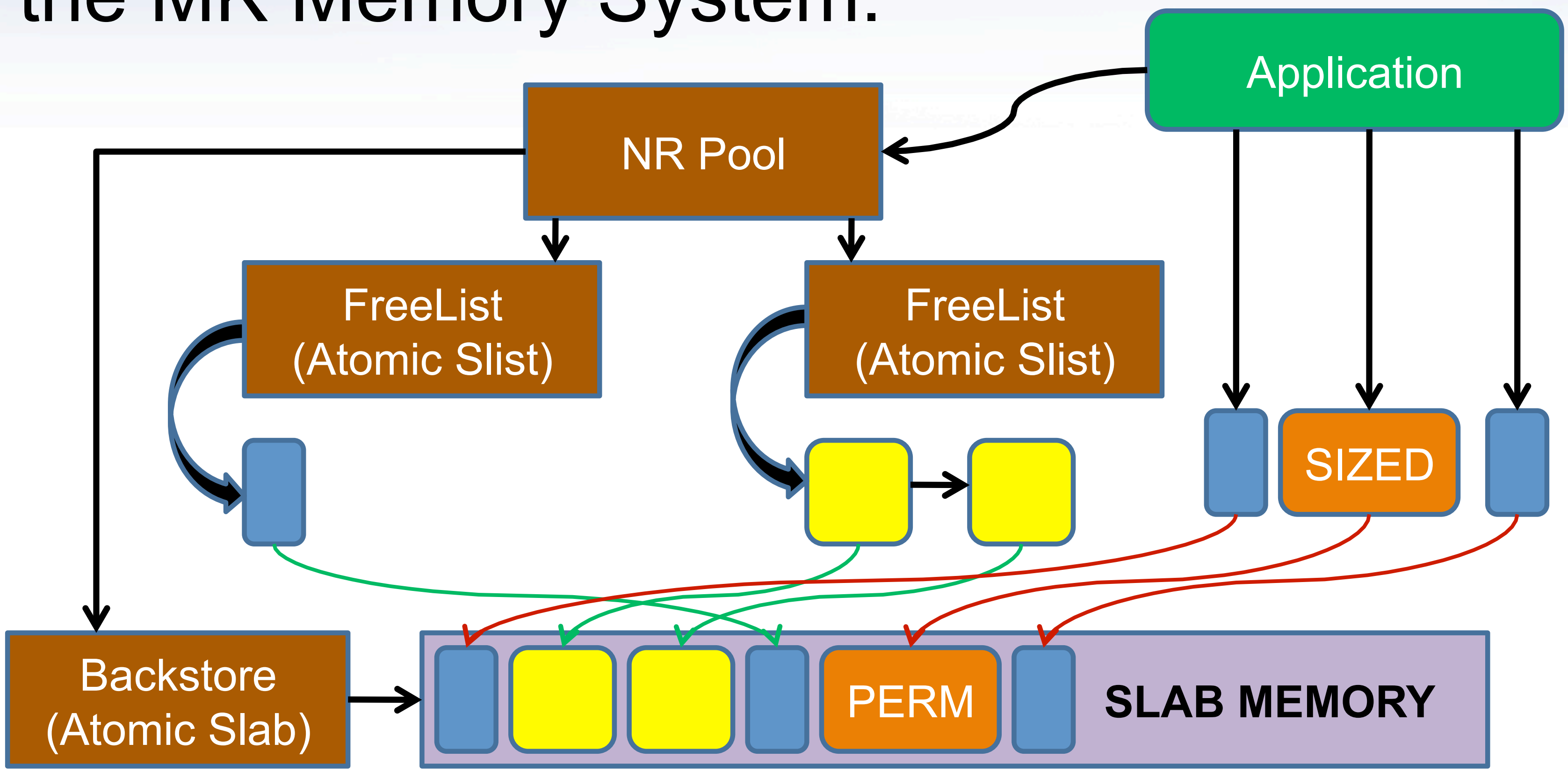
- AtomicPair is your friend. Allows you to access a pair of words atomically (read / write / CAS)
- Useful for a making a whole class of simple allocators LockFree and Multicore friendly
  - All allocators which use only two variable updates for control words
    - Concurrent FreeList (SLIST) [Head / ABA-Sequence]
    - Slab Allocator [Write-Pointer / Remainder]
    - Ring-Buffer [Read-Pointer / Write-Pointer] *

SLIST

- SLIST is a LockFree Singly-Linked List
  - Implemented in the Windows API
  - Very simple to roll your own (it's a good "hello world" for teaching LockFree programming)
  - Clever trick: Incorporate counter into ABA-Sequence for "free"
    - Example 32-bit Sequence starts at 0
    - Add 0x00010001 for Push
    - Add 0x0000FFFF for Pop
    - Bottom 16 bits == item count (up to 64K)

LockFree NR-Pools

Used for simple control structures in the MK Memory System.



Debugging Support

- Heap Validation Functions
- Memory Pattern Support (0xDEADBEEF et al)
- Basic Statistic Gathering
- Debug builds have extra heap integrity checks
- Debug Tracking can record all allocations
  – Exported to a file automatically on Out-of-Mem
  – Can track by specified “bins” or timed bread-crumbs
- Memory visualization tool: allocs & stack traces

Initialization Order

- Memory system must be initialized before C++ global constructors run if they call “new”.
- Construct-on-First-Use (COFU) has penalties for both implicit and explicit versions.
- Use Early-Init instead:

GCC:	__attribute__ ((init_priority (N)))
MSVC:	#pragma init_seg(X)

What went wrong…

- Underestimating amount of work
  - 10 months development prior to “live” deployment
  - 3 months up front writing support libraries alone
- Initial attempts at SBMM table sizing
  - Powers of 2 and Sparse Tables wasted memory
- Debug features had unclear messages
  - Asserts to trap memory corruption conditions led to many “crash in the memory system” reports that were flaws in game code

What went right

- Overall architecture
  - 3 Level Hybrid Heap approach for main allocator
- Building a library of multicore primitives
  - Now used by Rendering and Job Graph as well
- Building in additional debugging features
- Fairly easy to share with other projects
  - Example: 4 days to integrate without help
- Overall we are very pleased with the new system

Questions ???

Contact Info:

Adisak Pochanayon
Principal Software Engineer
Netherrealm Studios
adisak@wbgames.com

NETHERREALM STUDIOS