



# Dragged Kicking and Screaming: Source Multicore

Tom Leonard, Valve  
9 March 2007





# Multicore

- ③ Most significant development since consumer 3D





# Multicore

- ④ Most significant development since consumer 3D
- ④ Explicit parallelism
  - ④ Hardware problem becoming software problem will require new techniques





# Introduction

- ③ The decisions faced with multiple cores
- ③ How we are approaching multiple cores
- ③ Algorithms and paradigms



# Goals

- ⊕ Integrate multicore across Valve's business
  - ⊕ Expose to game programmers, licensees and MOD authors





# Goals

- ⊕ Integrate multicore across Valve's business
- ⊕ Scale to cores without recompile





# Goals

- ⊕ Integrate multicore across Valve's business
- ⊕ Scale to cores without recompile
- ⊕ Create value beyond framerate
  - ⊕ Apply cores to new gameplay



# Challenges

- ⊕ Games want maximal CPU utilization
- ⊕ Games are inherently serial
- ⊕ Decades of experience in single threaded optimization
- ⊕ Millions of lines of code written for single threading





# Strategies

- ③ Threading model
- ③ Threading framework





# Threading Models

- ④ Fine grained threading
- ④ Coarse threading
- ④ Hybrid threading



# Diving In

- ⊕ Client
  - ⊕ User input
  - ⊕ Rendering
  - ⊕ Graphics simulation
- ⊕ Server
  - ⊕ AI
  - ⊕ Physics
  - ⊕ Game logic





# Diving In

- ⊕ Experiment: run client and server each on own core





# Diving In

- ⊕ Experiment: run client and server each on own core
- ⊕ Benefits: forced to confront systems that are not thread safe or not thread efficient



# Discoveries

- ⊕ Problem: shared data access
  - ⊕ Global data
  - ⊕ Static data (optimizations/function local state)
  - ⊕ Singleton objects



# Discoveries

- ⊕ Problem: shared data access
- ⊕ Thread safety is easy!



# Discoveries

- ⊕ Problem: shared data access
- ⊕ Thread safety is easy!
  - ⊕ Slap on a mutex/critical section





# Discoveries

- ④ Problem: shared data access
- ④ *Bad* thread safety is easy!
  - ④ Slap on a mutex/critical section
  - ④ The simple thing is the worst thing
    - ④ Mutexes are terrible
      - ④ Excessive waits
      - ④ Error prone
      - ④ Fail to scale
  - ④ Establish slow but stable baseline



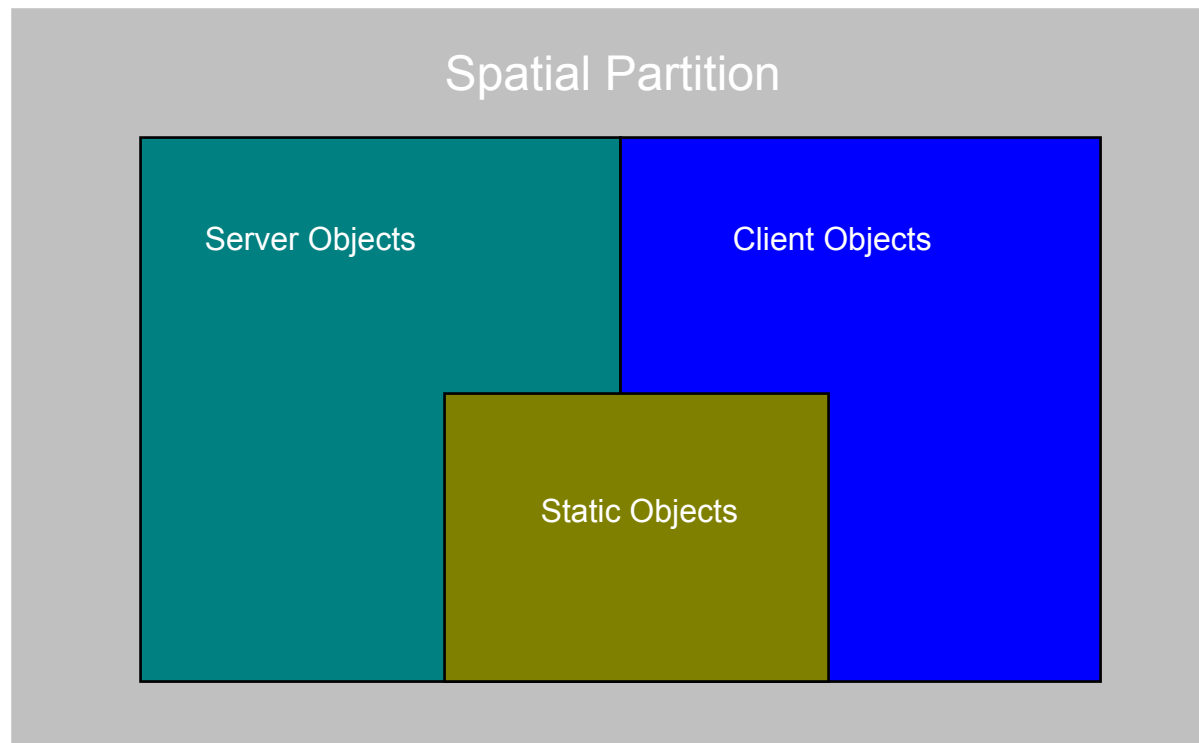


# Discoveries

- ③ Efficient thread safety
  - ③ No synchronization (“wait-free”)
    - ③ Each thread has a private copy of all the data needed to perform operation:
      - ③ Threads working on independent problems
      - ③ Replace globals with thread private data
      - ③ Reorient to pipeline
    - ③ Example: Source “Spatial Partition”

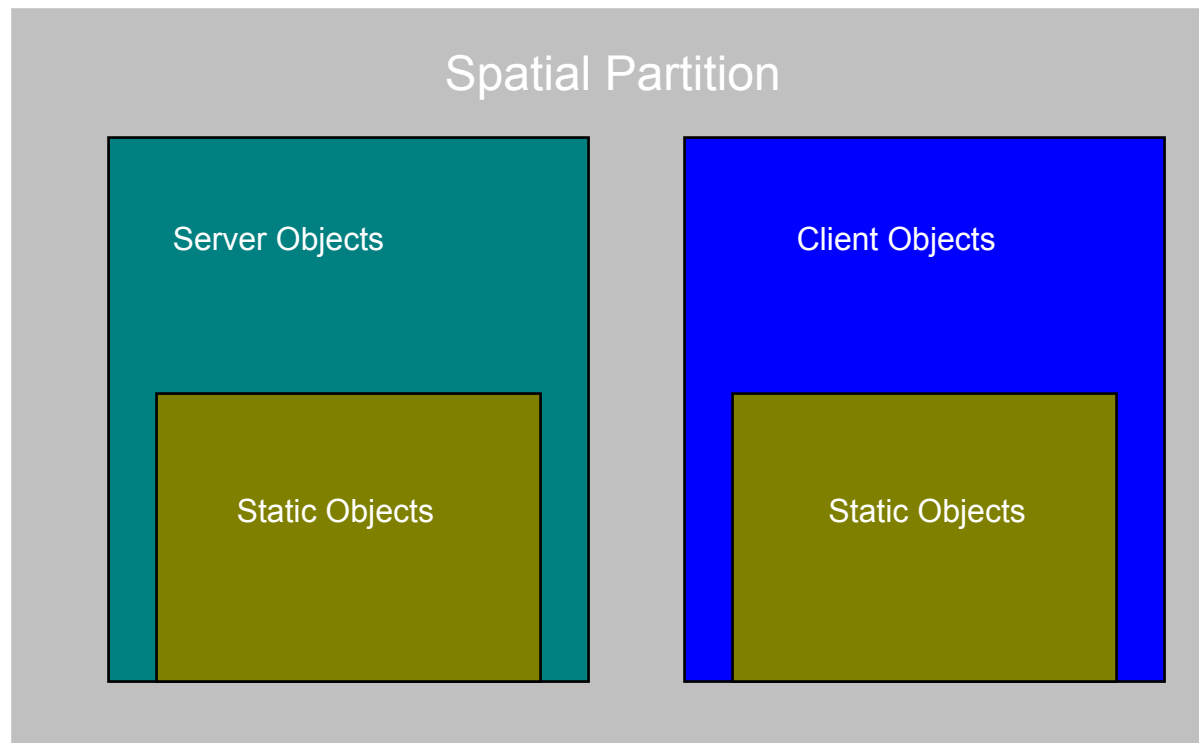


# Discoveries





# Discoveries





# Discoveries

- ③ Efficient thread safety
  - ③ No synchronization (“wait-free”)
  - ③ Better synchronization tools, techniques
    - ③ Analyze data access
    - ③ Example: symbol table using read/write lock
  - ③ Decouple using queued function calls



# Discoveries

- ③ What if you can't eliminate contention over shared resources?



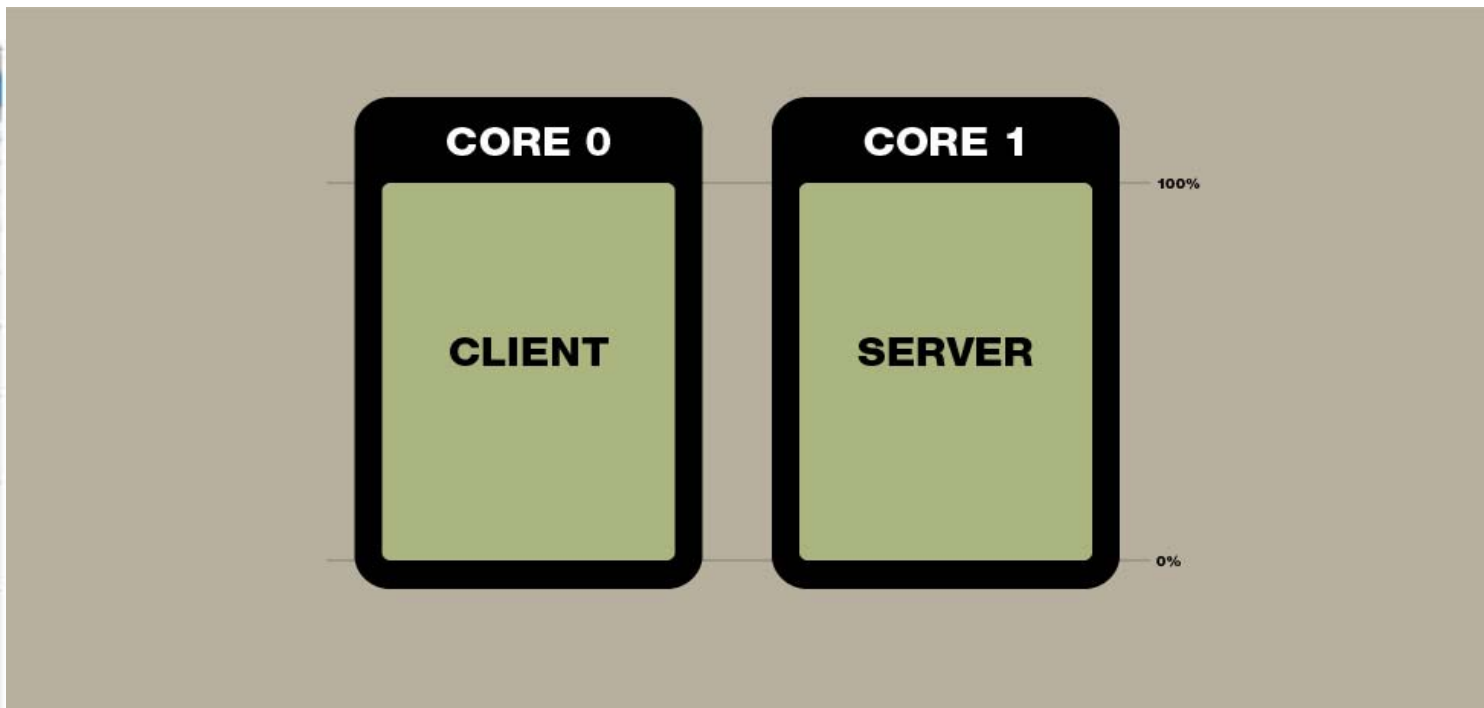


# Results

- ③ Can approach 2x in contrived maps



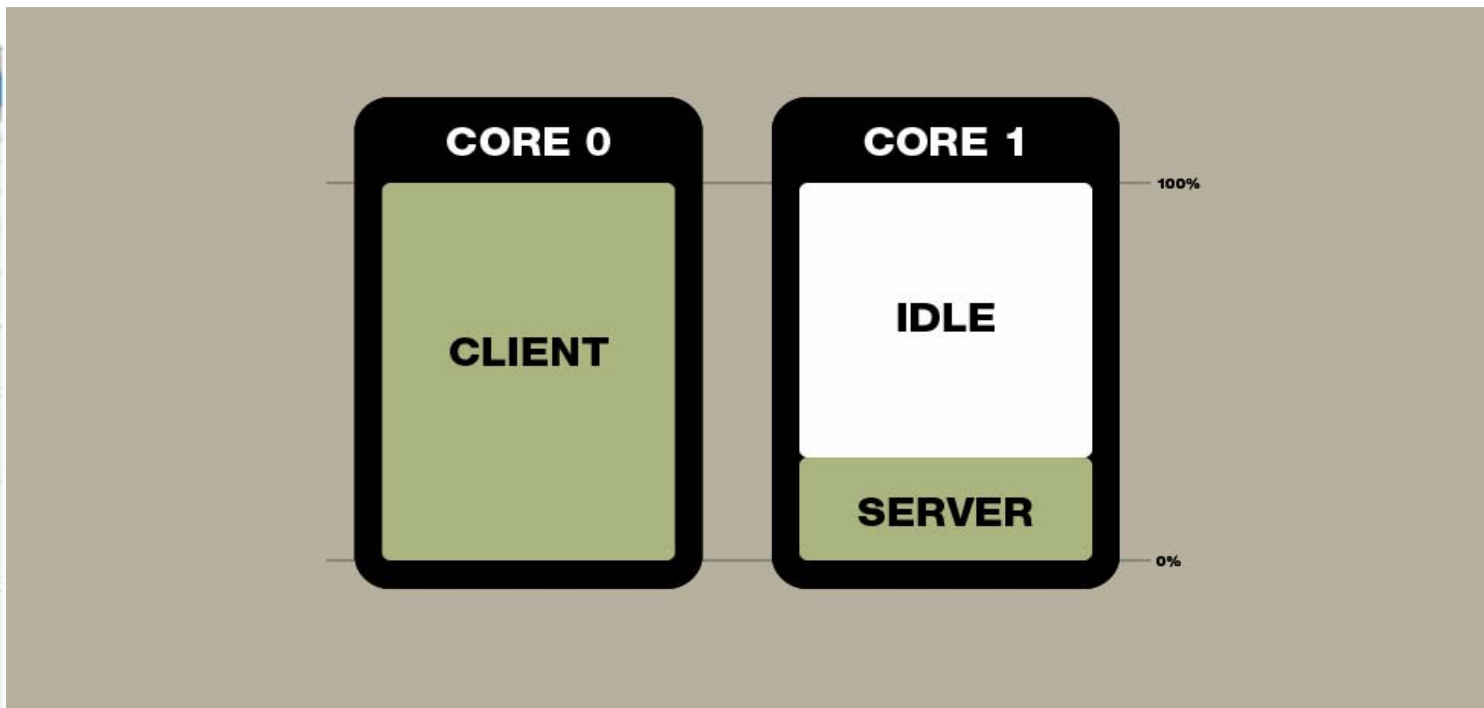
# Results







# Results





# Results

- ⊕ Can approach 2x in contrived maps
- ⊕ More like 1.2x in real single player
- ⊕ Applicable to 360 Team Fortress 2

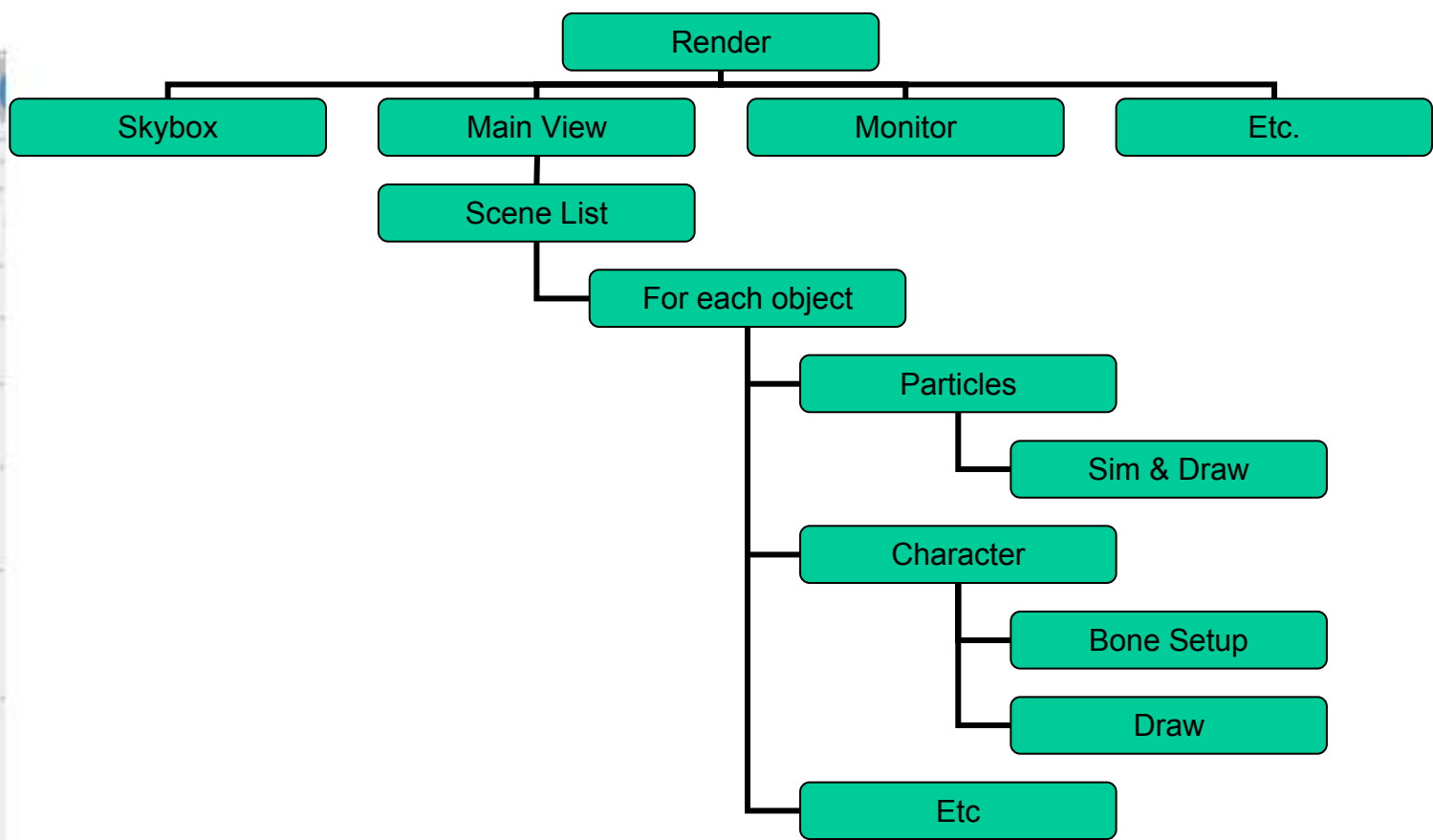


# Hybrid threading

- ④ Use the appropriate tool for the job
  - ④ Some systems on cores (e.g. sound)
  - ④ Some systems split internally in a coarse manner
  - ④ Split expensive iterations across cores fine grained
  - ④ Queue some work to run when a core goes idle
- ④ Need strong tools
- ④ Maximal core utilization



# Hybrid threading: Rendering

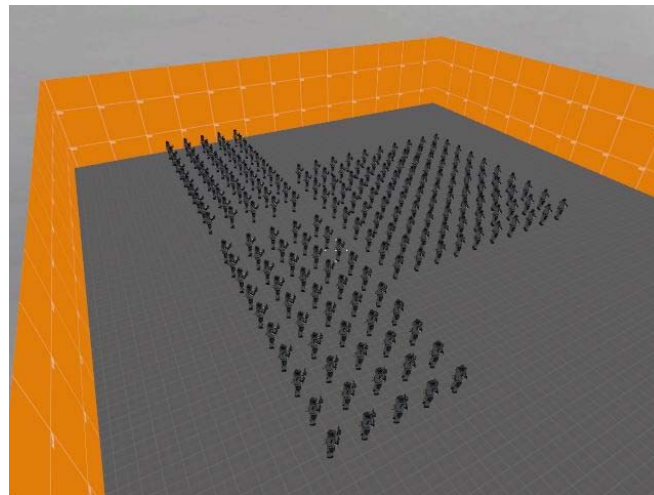




# Hybrid threading: Rendering

- ⊕ Problems
  - ⊕ Per-view scene construction limits opportunity
  - ⊕ Arbitrary object type order
  - ⊕ Arbitrary code execution
- ⊕ Simulation and Rendering interleaved
  - ⊕ Lazy calculation optimizations

# Hybrid threading: Rendering



- ④ Iterative Transition: Skeletal Animation
  - ④ Parallelize lazy calculation triggers
  - ④ Refactor bone setup into single pass per view
  - ④ Refactor into single pass for all views
  - ④ Same pattern for other CPU-intensive stages



# Hybrid threading: Rendering

- ⊕ Revised pipeline
  - ⊕ Construct scene rendering lists for multiple scenes in parallel (e.g., the world and its reflection in water)
  - ⊕ Overlap graphics simulation
  - ⊕ Compute character bone transformations for all characters in all scenes in parallel
  - ⊕ Allow multiple threads to draw in parallel
  - ⊕ Serialize drawing operations on another core



# Threading Tools

- ④ Implementing Hybrid Threading
- ④ Programmers solve game development problems, not threading problems
- ④ Empower all programmers to leverage cores
- ④ Operating system: too low level
- ④ Compiler extensions (OpenMP): too opaque
- ④ Tailored tools: correct abstraction





# Tailored tools: Game Threading Infrastructure

- ④ Custom work management system
  - ④ Intuitive for programmers
  - ④ Focus on keeping cores busy
  - ④ Thread pool: N-1 threads for N cores
  - ④ Support hybrid threading
    - ④ Function threading
    - ④ Array parallelism
    - ④ Queued and immediate execution



# Tailored tools: Game Threading Infrastructure

- ⊕ Goal: make system easy to use, hard to mess up
- ⊕ Example: compiler generated functors
  - ⊕ Uses templates to package up functions and data, point of call looks very similar
  - ⊕ Call arrives on other end as if called normally
  - ⊕ Saves time, reduces error, encourages experimentation



# Tailored tools: Game Threading Infrastructure

- ④ One-off push to another core

```
if ( !IsEngineThreaded() )  
    _Host_RunFrame_Server( numticks );  
else  
    ThreadExecute( _Host_RunFrame_Server, numticks );
```



# Tailored tools: Game Threading Infrastructure

## ③ Parallel loop

```
void ProcessPSystem( CParticleEffect *pEffect );
```

```
ParallelProcess( particlesToSimulate.Base(),  
                particlesToSimulate.Count(),  
                ProcessPSystem );
```



# Tailored tools: Game Threading Infrastructure

- ④ Queue up a bunch of work items, wait for them to complete

```
BeginExecuteParallel();  
ExecuteParallel( g_pParticleSystem,  
                &CParticleSystem::Update, time );  
ExecuteParallel( &UpdateRopes, time );  
EndExecuteParallel();
```

- ④ Low level APIs for the brave





# Contention

- ③ What if you can't eliminate contention over shared resources?
- ③ Example: Allocator
  - ③ Heavily used
  - ③ Multiple pools of fixed sized blocks with a custom spin lock mutex per-pool
  - ③ Mutex limiting scale
  - ③ Didn't want per-thread allocators



# Contention

- ⊕ Lock-free algorithms
  - ⊕ No thread can block system regardless of scheduling or state
  - ⊕ Under the hood of all services and data structures
  - ⊕ Relies on atomic write instructions, “compare-and-swap”



# Contention

```
bool CompareAndSwap(int *pDest, int newValue, int oldValue)
{
    Lock( pDest );
    bool success = false;
    if ( *pDest == oldValue )
    {
        *pDest = newValue;
        success = true;
    }
    Unlock( pDest );
    return success;
}
```





# Contention

```
bool CompareAndSwap(int *pDest, int newValue, int oldValue)
{
    __asm
    {
        mov eax,oldValue
        mov ecx,pDest
        mov edx,newValue
        lock cmpxchg [ecx],edx
        mov eax,0
        setz al
    }
}
```



# Contention

- ③ Use lock-free algorithm in allocator
  - ③ Replace mutex and traditional free list per-pool with a lock-free list per-pool
  - ③ Windows API/XDK SList



# Lock-free example: singly linked list

- ⊕ Compare-and-swap
  - ⊕ “If head is equal to what I think it is, assign with my new head”
  - ⊕ ABA Problem: is it the same head?
  - ⊕ Use a serial number as a discriminating field



# Lock-free example: singly linked list

```
class CSList
{
public:
    CSList()
    void Push( SListNode_t *pNode );
    SListNode_t *Pop();
    SListNode_t *Detach();
    int Count() const;
private:
    SListHead_t m_Head;
};
```



# Lock-free example: singly linked list

```
struct SListNode_t
{
    SListNode_t *pNext;
};

union SListHead_t
{
    struct value_t
    {
        SListNode_t *pNext;
        int16 iDepth;
        int16 iSequence;
    } value;
    int64 value64;
};
```





# Lock-free example: singly linked list

```
void Push( SListNode_t *pNode )
{
    SListHead_t oldHead, newHead;
    for (;;)
    {
        oldHead.value64 = m_Head.value64;
        newHead.value.iDepth = oldHead.value.iDepth + 1;
        newHead.value.iSequence = oldHead.value.iSequence + 1;
        newHead.value.Next = pNode;
        pNode->pNext = oldHead.value.pNext;
        if ( ThreadInterlockedAssignIf64( &m_Head.value64,
            newHead.value64, oldHead.value64 ) )
        {
            return;
        }
    }
}
```





# Lock-free example: singly linked list

- ③ Lock-free list exceptionally useful
  - ③ Keep pools of context structures when impractical to give every thread a context
  - ③ Efficiently gather results of a parallel process for later handling
  - ③ Build up lists of data to operate on using `Push()`, then use `Detach()` (a.k.a “Flush”) to grab the data in another thread in a single operation



# Example

```
extern Vector trace_start;
extern Vector trace_end;
// etc...
struct cbrush_t
{
    int          contents;
    unsigned short numsides;
    unsigned short firstbrushside;
    int          checkcount; // to avoid repeated testings
};

////////////////////////////////////

void BeginTrace()
{
    g_CModelMutex.Lock();
    ++s_nCheckCount;
}
```







# Example

```
struct TraceInfo_t
{
    Vector m_start;
    Vector m_end;
    // etc...
    CVisitBitVec m_BrushVisits;
};

CTraceInfoPool g_TraceInfoPool;

TraceInfo_t *BeginTrace()
{
    TraceInfo_t *pTraceInfo;
    if ( !g_TraceInfoPool.PopItem( &pTraceInfo ) )
        pTraceInfo = new TraceInfo_t;

    return pTraceInfo;
}
```





# Lock-free algorithms

- ③ Thread pool work distribution queue
  - ③ Derived from HL2 asynchronous I/O queue
  - ③ Designed for one provider, one consumer
  - ③ Simple prioritized queue with mutex
  - ③ Arbitrary priority
  - ③ One queue for all threads



# Lock-free algorithms

## ④ Solutions

- ④ Use lock-free queue (Fober, et. al.)
- ④ Rework interface to fixed priorities, one queue per-priority
  - ④ *Interfaces critical*
- ④ Queues per core in addition to a shared queue
- ④ Use atomic operations to get “ticket”, actual work done may differ



# Lock-free algorithms

- ⊕ Locks permit a stable reality
- ⊕ Lock-free permits reality to change instruction to instruction
- ⊕ Leverage inference rather than locks to know part of the system is stable
- ⊕ Wait-free is always better



# Looking Forward

- ④ Why so much up-front investment?





# Looking Forward

- ④ Why so much up-front investment?
  - ④ Steam
    - ④ Communicate with customers
    - ④ Tap markets not available via retail
  - ④ Dramatic change is underway
    - ④ Core count double every 18 months
    - ④ CPU/GPU/PPU/AIPU/etc not the future
    - ④ Many homogeneous cores
    - ④ Division of computing power a software problem



# Call to action

- ④ Build or acquire strong tools, new techniques
- ④ Embrace lock-free mechanisms to move work and data to and from wait-free code
- ④ Prepare for decomposition of features over many cores
- ④ Use accessible solutions to empower all programmers, not just systems programmers
- ④ Support even higher level threading framed in terms of game problems





# Summary

- ⊕ Started with a stable but bad threading
- ⊕ Iteratively eliminated bad cases using variety of techniques, usually lock-free
- ⊕ During iterations, expanded toolset to meet newly discovered needs
- ⊕ Focused on ease-of-use for other programmers
- ⊕ Now being applied by others at higher levels





⊕ In Source SDK this summer

⊕ Contact: [tom\\_gdc@valvesoftware.com](mailto:tom_gdc@valvesoftware.com)

