Multi-threaded Optimization by Example

Introduction to minimizing communication barriers

(Mike Acton)

Peeling Grapes with a Sledgehammer

- General-Purpose Options
  - Spinlock – Block until value can be changed.
  - Semaphore – Block if no remaining items
  - Mutex – Block if already locked
  - Conditional Variables – Block for event signal
  - Critical Section – Same as mutex. (Sometimes lighter weight.)
  - ...and there are more

The malloc() of multi-threading.

- We aren't solving general-purpose problems.
- We're solving problems specific to our games and our tech.
- "Thread-safe" is a completely useless phrase in this context.

BACKGROUND

Understand the basics for the PPU

Spinlocks

- Fundamental
- Hardware backed
- OS Independent

Spinlocks

- LWARX (load word and reserve indexed)
- STWCX (store word conditional indexed)

```
atomic_increment( int32_t* addr )

retry:
   lwarx  r4, 0,  r3     ; Load the value. Reserve cacheline.
   addi   r4, r4, 0x01   ; Increment loaded value
   stwcx. r4, 0,  r3     ; Try to store value.
   bne- .retry           ; Reservation lost, try again.
```

The Bad

- Work on full cacheline (128 bytes)
- Only one reservation per processor (i.e. A total of ONE for us.)
  - The more spinlocks you use, the less effective they are.
  - ...and every other synchronization primitive will use some variation of them.

The Good

- Relatively low overhead
- Straightforward and simple
- Usable in userspace

OS-Level Synchronization Primitives

- No “hard” definitions. Vary with OS.
- Each are (more or less) functionally equivalent.
- Imply at least wait_event and set_event system calls.
- Look at examples:
  - Mutex
  - Semaphore

Mutex

- AKA Binary Semaphore (w/ restrictions)
- API: Create(), Destroy(), Lock(), Unlock()
- Typical rules:
  – Only the locking thread can unlock
  – No recursive locking
  – No multiple unlocks
  – No multiple initialization
  – Thread cannot exit with open mutexes
  – Cannot use during interrupt callback

Mutex (Lock)

```
// NOTE: pthread code referenced in these slides has
// been simplified somewhat to fit.

int pthread_mutex_lock(pthread_mutex_t *mutex)
{
  // Is the mutex unlocked already?
  if (atomic_exchange(&mutex->lock, 1) != 0)
  {
    // Stall until we can get a lock.
    while (atomic_exchange(&mutex->lock, -1) != 0)
    {
      // Add to OS thread queue
      if (wait_event(mutex->event, INFINITE) != 0)
          return EINVAL;
    }
  }
  return (0);
}
```

atomic_exchange

- Simplest use of lwarx and stwcx
- Returns previous value

```
atomic_exchange( int32_t* addr, int32_t value )

retry:
   lwarx  r0, 0,  r3    ; Load the value. Reserve cacheline.
   stwcx. r4, 0,  r3    ; Try to store value.
   bne- .retry          ; Reservation lost, try again.
   mr     r3, r0        ; Copy previous value to return
```

Mutex (Unlock)

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
{
    idx = atomic_exchange(&mutex->lock, 0);
    // Is the mutex locked?
    if (idx != 0)
    {
      // A lock is currently waiting for this event.
      if (idx < 0)
      {
        if (set_event(mutex->event) < 0)
          return EINVAL;
      }
    }
    else
      return EPERM;

    return (0);
}
```

Semaphore

- AKA Counting Semaphore
- API: Create(), Destroy(), Get(), Up(), Down()
- Typical rules:
  - No owner
  - Can set value on creation
  - Cannot reset value directly
  - Cannot read value

Semaphore (Counting)

```
// The basis of a semaphore is being able to count
// -1 is interpreted as ZERO

int sem_update_count(int32_t* sem_count, int incr);

.retry:
     lwarx   r0,0, r3 ; Load the value. Reserve cacheline.
     srawi   r1,r0,31 ; r1 = (r0 < 0)?0xffffffff:0
     andc    r1,r0,r1 ; r1 = r0 & ~r1 (Zero if negative)
     add     r1,r1,r4 ; r1 = r1 + incr
     stwcx.  r1,0, r3 ; Try to store new value
     bne     .retry   ; Reservation lost, try again.
     mr      r3, r0   ; Copy previous value to return
```

Semaphore (Basic API)

```
// NOTE: Grossly over-simplified to fit on slide.

int sem_up( semaphore* sem )
{
   sem_update_count( &sem->count, +1 );
   set_event( &sem->event );
   return (0);
}

int sem_down( semaphore* sem )
{
   // Wait until an item is available
   while ( sem_update_count( sem_count, -1 ) == 0 )
   {
     wait_event( &sem->event );
   }
   return (0);
}
```

Typical Example

- Fixed size buffer of objects (queue)
- Thread-1 (Producer)
  - Adds objects to queue so long as space is available.
- Thread-2 (Consumer)
  - Removes (processes) objects from queue so long as there are objects remaining.

Example (1) Producer

```
Init:
  mutex_init( &full_of_ideas, LOCKED )
  mutex_init( &out_of_ideas, LOCKED )

while (1)
{
  if ( queue_is_full( &idea_queue ) )
  {
      mutex_unlock( &full_of_ideas );
      mutex_lock( &out_of_ideas );
  }
  idea = make_idea();
  mutex_lock( &idea_key );
  queue_push( &idea_queue, idea );
  mutex_unlock( &idea_key );
}
```

Example (1) Consumer

```
while (1)
{
  mutex_lock( &full_of_ideas );

  mutex_lock( &idea_key );
  while ( queue_is_not_empty( &idea_queue ) )
  {
     idea = queue_pop( &idea_queue );
     take_advantage_of( &idea );
  }
  mutex_unlock( &idea_key );

  mutex_unlock( &out_of_ideas );
}
```

Example (2) Producer

```
Init:
  semaphore_init( &ideas_made, QUEUE_SIZE );
  semaphore_init( &ideas_used, 0 );
  semaphore_init( &idea_key,   1 );

while (1)
{
  semaphore_down( &ideas_made );
  idea = make_idea();
  semaphore_down( &idea_key );
  queue_push( &idea_queue, idea );
  semaphore_up( &idea_key );
  semaphore_up( &ideas_used );
}
```

Example (2) Consumer

```
while (1)
{
  semaphore_down( &ideas_used );
  semaphore_down( &idea_key );
  idea = queue_pop( &idea_queue );
  semaphore_up( &idea_key );
  take_advantage_of( &idea );
  semaphore_up( &ideas_made );
}
```

End of background section

Now let's talk about where it can be done better.

REAL EXAMPLE

Look at the actual queue used by the TcpServer example (later)

Atomic Read/Write

- Unfortunately “atomic” has been too heavily overloaded to use meaningfully.
- Typically, at a high level: “Atomic” means both indivisible and guaranteed. e.g. spinlock atomic_increment().
- But it also simply means indivisible. i.e. It will either succeed or not, but never partially.

Atomic Functions

```
// These functions must not be inlined on the x86. In order to ensure that
// the writes and reads are atomic, the compiler must not be given
// any opportunity to fold the operation into a more complex instruction
// e.g. load-increment-store

void     AtomicWrite32Aligned( uintptr_t address, uint32_t arg );
void     AtomicWritePointerAligned( uintptr_t address, uintptr_t arg );
uint32_t AtomicRead32Aligned( uintptr_t address );
```

- Why is the x86 different from the PPC here?

Side-Topic (Validation)

- What if a read or write needs to be atomic but it wasn’t?
  - Multi-threaded programming and optimization are by-contract (mutual agreement)
  - Similar to data (parameter) contracts to functions, but…
  - There is often no way to meaningfully validate correctness (assert).
  - Be thorough. Test. Talk it through with someone.

Atomic Functions

```
void
AtomicWrite32Aligned( uintptr_t address, uint32_t arg )
{
    // Aligned 32 bit writes are always atomic on x86
    // Aligned 32 bit writes are always atomic on PowerPC

    assert( (address & 0x03) == 0 );

    uint32_t* p = (uint32_t*)address;

    *p = arg;
}
```

- Could this, in fact, be inlined?
  - Yes – with inline assembly to select the proper instruction.

- BUT LOOK OUT!
  - Peephole optimizations
  - Link-level optimizations

Atomic Functions

```
void
AtomicWritePointerAligned( uintptr_t address, uintptr_t arg )
{
    // Aligned 32 bit writes are always atomic on x86
    // Aligned 32 bit writes are always atomic on PowerPC

    // Aligned 64 bit writes are not always atomic on x86, so if
    // pointers are 64 bits, this needs to be re-written.

    assert( (address & 0x03) == 0 );

    uintptr_t* p = (uintptr_t*)address;

    *p = arg;
}
```

Atomic Functions

```
uint32_t
AtomicRead32Aligned( uintptr_t address )
{
    // Aligned 32 bit reads are always atomic on x86
    // Aligned 32 bit reads are always atomic on PowerPC

    assert( (address & 0x03) == 0 );

    volatile uint32_t* p = (uint32_t*)address;

    return (*p);
}
```

- NOTE: Use of volatile is not strictly necessary here. There is no way for this read to get folded out.

- On the PPC this same function could be inlined and the volatile would be necessary.

Fixed Queue Example

- Some things are knowable – this isn't a theoretical exersize:
  - Very Common Case:
    - Single Reader (Consumer)
    - Single Writer (Producer)
  - Plan the data access patterns.
    - Multithreading is much easier for read-only and write-only cases.
    - Use read-write carefully.

Fixed Queue Example

Single-Reader, Single-Writer
Fixed-Sized Lookaside Dequeue

Rules for use:

- Push() can only be called from one thread, but it doesn't need to be the same thread as Pop().
- Pop() can only be called from one thread, but it doesn't need to be the same thread as Push().
- max_count must be a power of two.
- Don't Pop() more than Count()
- Check IsFull() or Count() before a Push()

SrSwFixedLookasideDequeue

```
struct SrSwFixedLookasideDequeue
{
    // DO NOT MODIFY DIRECTLY
    uint32_t  m_pushed_count;
    uint32_t  m_popped_count;
    uint32_t  m_max_count;

    // API
    void      Clear( uint32_t max_count );
    uint32_t  Count();
    int       IsFull();
    uint32_t  NextPushElement();
    void      Push();
    uint32_t  NextPopElement();
    void      Pop();
};
```

SrSwFixedLookasideDequeue

```
inline void
SrSwFixedLookasideDequeue::Clear( uint32_t max_count )
{
  // These writes don't need to be atomic. The assumption is that either:
  //
  //     (1) This is only done once before the queue is actually used
  //         (i.e. init), or
  //     (2) If the queue is going to be cleared there will certainly be some
  //         need to synchronize at a higher level, so pointless to do it here
  //         too.

  m_pushed_count = 0;
  m_popped_count = 0;
  m_max_count    = max_count;
}
```

SrSwFixedLookasideDequeue

```
// Get Number of items in the queue

inline uint32_t
SrSwFixedLookasideDequeue::Count()
{
    uintptr_t pushed_count_addr = (uintptr_t)&m_pushed_count;
    uintptr_t popped_count_addr = (uintptr_t)&m_popped_count;
    uint32_t  pushed_count      = AtomicRead32Aligned( pushed_count_addr );
    uint32_t  popped_count      = AtomicRead32Aligned( popped_count_addr );
    uint32_t  diff              = pushed_count - popped_count;

    return (diff);
}
```

- These reads will be complete (atomic) but not necessarily most-recent. (This is not a problem.)

- NOTE: Because counts are unsigned, overflow is not a problem.
    - e.g. in 8 bits: 0x01 (1) – 0xfe (254) = 0x03 (3)

SrSwFixedLookasideDequeue

```
// Check if the queue is full.

inline int
SrSwFixedLookasideDequeue::IsFull()
{
    uint32_t element_count = Count();

    return (element_count == m_max_count);
}
```

SrSwFixedLookasideDequeue

- With SrSw it doesn't matter if Count() changes while or immediately after IsFull() is called:
  - If IsFull() is used to determine if there is space –
    - it is the thread that does the Push() that makes this call so…
    - it cannot be made more full by another thread,
    - only more empty.
  - If IsFull() is called by the Pop()'ing thread (Consumer)
    - (If TRUE) Only this thread can make it less full, so it is necessarily true.
    - (If FALSE) If can be made full immediately after, and the check can be missed – but since this can only be conceivably used for polling – it will be hit the next time around.

SrSwFixedLookasideDequeue

```
// Return the index of the next element that would be pushed.

inline uint32_t
SrSwFixedLookasideDequeue::NextPushElement()
{
    uintptr_t pushed_count_addr = (uintptr_t)&m_pushed_count;
    uint32_t  pushed_count      = AtomicRead32Aligned( pushed_count_addr );
    uint32_t  element_ndx       = pushed_count & ( m_max_count - 1 );

    return (element_ndx);
}
```

- This function can only be called from the Push()'ing thread (Producer)
- The count read is always the complete (atomic) and the most recent (only this thread can change it.)
- element_ndx = pushed_count % m_max_count;

SrSwFixedLookasideDequeue

```
inline void
SrSwFixedLookasideDequeue::Push()
{
    uintptr_t pushed_count_addr = (uintptr_t)&m_pushed_count;
    uint32_t  pushed_count      = AtomicRead32Aligned( pushed_count_addr ) + 1;

    AtomicWrite32Aligned( pushed_count_addr, pushed_count );
}
```

- The incremement itself does not need to be atomic since only a single thread can write to this address.
- But the write must be atomic since multiple threads can read.

SrSwFixedLookasideDequeue

```
// Return the index of the next element that would be popped.

inline uint32_t
SrSwFixedLookasideDequeue::NextPopElement()
{
    uintptr_t popped_count_addr = (uintptr_t)&m_popped_count;
    uint32_t  popped_count      = AtomicRead32Aligned( popped_count_addr );
    uint32_t  element_ndx       = popped_count & ( m_max_count - 1 );

    return (element_ndx);
}
```

- Pretty much the same as NextPushElement()
- This function can only be called from the Pop()'ing thread (Consumer)
- The count read is always the complete (atomic) and the most recent (only this thread can change it.)
- element_ndx = popped_count % m_max_count;

SrSwFixedLookasideDequeue

```
inline void
SrSwFixedLookasideDequeue::Pop()
{
    uintptr_t popped_count_addr = (uintptr_t)&m_popped_count;
    uint32_t  popped_count      = AtomicRead32Aligned( popped_count_addr ) + 1;

    AtomicWrite32Aligned( popped_count_addr, popped_count );
}
```

- Pretty much the same as Push()
- The incremement itself does not need to be atomic since only a single thread can write to this address.
- But the write must be atomic since multiple threads can read.

SrSwFixedLookasideDequeue

- That's it.

- This is now a very simple, very fast queue that can be used in a multithreaded environment.

Example (3) Producer

```
Init:
  Idea idea_buffer[ QUEUE_SIZE ]; // QUEUE_SIZE=power of two
  SwSrFixedLookasideDequeue idea_queue;
  idea_queue.Clear( QUEUE_SIZE );

while (1)
{
  if ( idea_queue.IsFull() )
  {
     // Sleep, stall, fail or do some other work.
  }

  uint32_t next_idea_ndx = idea_queue.NextPushElement();
  Idea     idea          = make_idea();

  idea_buffer[ next_idea_ndx ] = idea;
  AtomicStoreFence();
  idea_queue.Push();
}
```

- Note that the buffer is written to before the push is done. If it is not done in this order, there is a race condition.
- If there is a store reorder buffer on the processor, a store fence must be used to guarantee the order.

Atomic Load/Store Fence

```
// These fences are only needed on architectures that have load-store re-ordering
// These only apply to access of system memory

// Pentium
#define  AtomicStoreFence() __asm { sfence }
#define  AtomicLoadFence()  __asm { lfence }

// PowerPC
#define  AtomicStoreFence() __asm { lwsync }
#define  AtomicLoadFence()  __asm { lwsync }

// But on the PPU
#define  AtomicStoreFence()
#define  AtomicLoadFence()
```

- For system memory accesses –
  - StoreFence: Ensure that all previous stores will be completed before any following stores.
  - LoadFence: ensure that all previous loads will be completed before any following loads.

PowerPC Sync

- For PowerPC, sometimes you will see { sync }
  - This is a “heavy-weight” sync – All load/store instructions before the sync will be completed before any instructions following.
  - Including instructions across:
    - System memory
    - Cache inhibited (uncached memory)
    - Gaurded (I/O mapped device memory e.g. SPU addresses, etc.)

- For PowerPC, sometimes you will see { eieio }
  - Enforce In-Order Execution of I/O
  - This is basically the same as a { sync } enforces the order within each type of memory, not across all types.

Example (3) Producer

```
while (1)
{
  if ( idea_queue.IsFull() )
  {
    // Sleep, stall, fail or do some other work.
  }

  . . .
```

- Where you would normally see a Sleep() in a standalone server, in the context of a game we have both a main loop and a vsync – natural places to “restart” background tasks.
- The frame can also be split into multiple sub-frames that trigger (and spread) the load across the game cycle.

```
  if ( idea_queue.IsFull() )
  {
    return; // Will add it next time around.
  }

  . . .
```

Example (3) Consumer

```
uint32_t idea_count = idea_queue.Count();
while (idea_count)
{
  uint32_t idea_ndx = idea_queue.NextPopElement();
  Idea     idea     = idea_buffer[ idea_ndx ];
  AtomicLoadFence();
  idea_queue.Pop();
  idea_count--;
}
```

- In the same way as the Producer, the data must be taken from the buffer before it is Pop()'d or there will be a race condition.

TCP SERVER

Putting some of this together toward a faster TCP server.

TcpServer

Concept:

(1) Be pretty close to the design ideas of the pre-existing one in IPC

(2) No standard sync primitives (e.g. compare-swap, critical-section, etc.) i.e. completely lock-free.

(3) Support multiple client connections.

TcpServer

```
#define TCPSERVER_MAX_CONNECTIONS 256
#define TCPSERVER_QUEUE_LENGTH    64

typedef void* (TcpServerAllocFunc) ( size_t size );
typedef void  (TcpServerFreeFunc)  ( void* ptr   );

// (1) A TcpServer must be aligned on a 32 bit boundary
// (2) A TcpServer can only be owned by a single user thread.

struct TcpServer
{
```

TcpServer

```
//
// NO DATA SHOULD BE MODIFIED DIRECTLY
//

// This data is only modifiable from the server thread. Non-atomic access
// is acceptable. This data may not be accessed by the user thread.

IPC::TcpConnection   m_connection;
uint16_t             m_server_port;
uint16_t             _pad_0;
Socket               m_server_socket;
ThreadHandle         m_server_thread;
int                  m_server_thread_running;

Socket               m_client_connection_table [TCPSERVER_MAX_CONNECTIONS];
uint32_t             m_client_addr_table       [TCPSERVER_MAX_CONNECTIONS];
uint32_t             m_client_connection_end_count;
```

Everything is 32 bit aligned.
Need to know the size of all the types.
This is made more difficult with extra opaque typing layers.

```
sizeof(Socket)       == sizeof(int)
sizeof(ThreadHandle) == sizeof(HANDLE)           == sizeof(void*)    // WIN32
sizeof(ThreadHandle) == sizeof(sys_ppu_thread_t) == sizeof(uint64_t) // PPU
```

TcpServer

```
// This data is only read-write from the server thread. This data is is
// read-only from the user thread. Reads and writes must be atomic.

int32_t                   m_server_thread_errno;
int32_t                   m_server_thread_errln;
```

Note with multithreaded design, there must be very clear rules for how data is accessed.

Otherwise, everything will get out of control and non-optimizable. (Remember malloc()!)

TcpServer

```
// The inbound message dequeue is read-write from the server thread and
// read-only from the user thread. Atomic access is handled by
// SrSwFixedLoookasideDequeue.

SrSwFixedLookasideDequeue  m_inbound_message_dequeue;

// The outbound message dequeue is read-write from the user thread and
// read-only from the server thread. Atomic access is handled by
// SrSwFixedLoookasideDequeue.

SrSwFixedLookasideDequeue  m_outbound_message_dequeue;

TcpServerMessage*          m_inbound_messages  [ TCPSERVER_QUEUE_LENGTH ];
TcpServerMessage*          m_outbound_messages [ TCPSERVER_QUEUE_LENGTH ];
```

Note that separate queues are used. This is based on the principle of separating read data and written data. This makes things much easier. (And faster)

This principle does not just apply to multithreading, either!

TcpServer

```
// These functions are callable from either thread, but can only be written
// once in initialization. The memory manager must be atomic across
// multiple threads because ownership is shifted from one thread to
// another.

TcpServerAllocFunc*         m_call_alloc;
TcpServerFreeFunc*          m_call_free;
```

Allocating messages was in the original design.

Assumption: There is some fast method of allocating messages in the calling context. e.g. a fixed pool of messages.

Assumption: This may be used in different contexts within the same application – so may need different alloc/free routines. So, not using static members.

TcpServer

```
//
// PUBLIC API (Called from user thread)
//

int                 Start( uint16_t port, TcpServerAllocFunc* alloc,
                                          TcpServerFreeFunc*  free );
void                WaitForClose();
void                Kill();
int32_t             GetErrno();
int32_t             GetErrln();

TcpServerMessage*   CreateMessage( uint32_t tag, uint32_t size,
                                   uint32_t transaction,
                                   uint32_t client_ndx,
                                   uint32_t client_addr );

void                DestroyMessage( TcpServerMessage* message );
uint32_t            CountInboundMessages();
uint32_t            CountOutboundMessages();
void                PopInboundMessage( TcpServerMessage** message_result );
int                 PushOutboundMessage( TcpServerMessage* message );
```

Taking a message from the system implies you take ownership of the data.
Giving a message to the system implied you give away ownership of the data.

TcpServer

```
    //
    // INTERNAL (Called from server thread)
    //

    void                  SetError( int32_t thread_errno, int32_t thread_errln );
    int                   PushInboundMessage( TcpServerMessage* message );
    void                  PopOutboundMessage( TcpServerMessage** message_result );
    int                   AcceptNewClient();
    int                   HandleIncomingData( fd_set* server_read_fd );
    void                  HandleOutgoingData();

    int                   ReadSocketData( uint32_t client_ndx, char* buffer,
                                          uint32_t buffer_size );

    int                   WriteSocketData( uint32_t client_ndx, char* buffer,
                                           uint32_t buffer_size );

    void                  CloseClientConnection( uint32_t client_ndx );
};
```

EchoServer

- First, look at EchoServer.cpp for example of how this is used.
- Then, look at the interesting parts of TcpServer.cpp

TcpServer (Internal)

```
void
TcpServer::SetError( int32_t thread_errno, int32_t thread_errln )
{
    uintptr_t thread_errno_addr = (uintptr_t)&m_server_thread_errno;
    uintptr_t thread_errln_addr = (uintptr_t)&m_server_thread_errln;

    // The write order is important. errln must be written first, so
    // that when errno != 0 is read, errln is gauranteed to be ready.

    AtomicWrite32Aligned( thread_errln_addr, (uint32_t)thread_errln );
    AtomicStoreFence();
    AtomicWrite32Aligned( thread_errno_addr, (uint32_t)thread_errno );
}
```

An error is made up of two parts – the errno and the line it occurred on.
We must ensure that they are read together.

This principle does not just apply to multithreading, either!

TcpServer (Internal)

```
void
TcpServer::PopInboundMessage( TcpServerMessage** message_result )
{
    uint32_t message_ndx = m_inbound_message_dequeue.NextPopElement();

    // The read from m_inbound_messages does not need to be atomic since
    // (1) the write was gauranteed to be complete (i.e. was atomic) and,
    // (2) the value cannot be over-written until the Pop() is complete.

    *message_result = m_inbound_messages[ message_ndx ];

    // But it is important that the pop be done after the pointer is read
    // i.e. ownership has been given to the user thread, because once the
    // pop is complete it may be immediately overwritten by the server thread.
    // Doing the pop before reading the pointer would cause a race condition.

    AtomicLoadFence();

    m_inbound_message_dequeue.Pop();
}
```

Pretty much the same as the Consumer example.
But note the non-atomic read.

TcpServer (Internal)

```
int
TcpServer::PushOutboundMessage( TcpServerMessage* outbound_message )
{
    if ( m_outbound_message_dequeue.IsFull() )
    {
        return (0);
    }

    uint32_t          message_ndx  =
                      m_outbound_message_dequeue.NextPushElement();

    TcpServerMessage** outbound_message_ind =
                      &m_outbound_messages[ message_ndx ];

    AtomicWritePointerAligned( (uintptr_t)outbound_message_ind,
                               (uintptr_t)outbound_message );

    AtomicStoreFence();

    m_outbound_message_dequeue.Push();

    return (1);
}
```

Pretty much the same as the Producer example.
But note atomic write of the pointer, which will be required by the pop.

TcpServer (Internal)

- HandleOutgoingData() processes all the pending messages sends them to the appropriate clients.

```
void
TcpServer::HandleOutgoingData()
{
    uint32_t outbound_message_count = CountOutboundMessages();

    while ( outbound_message_count )
    {
```

- Note, the count is only read once. The server only empties out the part of the queue it knows about at the start.
- New messages may be added while these are being processed, no problem.
- They will just be processed next time around.

TcpServer (Internal)

```
TcpServerMessage*        h_outbound_message;
TcpClientMessageHeader n_outbound_message_header;

PopOutboundMessage( &h_outbound_message );

int32_t message_client_ndx  = h_outbound_message->m_header.m_client_ndx;
int32_t message_client_addr = h_outbound_message->m_header.m_client_addr;

Socket   client_socket        = m_client_connection_table[ message_client_ndx ];
uint32_t client_addr          = m_client_addr_table[ message_client_ndx ];

if ( client_socket == m_server_socket )
{
  // This connection has been dropped. There's no where for this message to go.
  // Drop message.
  goto CLEANUP_OUTBOUND_MESSAGE;
}

if ( client_addr != message_client_addr )
{
  // This connection has been dropped and since re-opened with a new client.
  // Drop message.
  goto CLEANUP_OUTBOUND_MESSAGE;
}
```

- Note Client and Server Message headers are different
– Server stores more information.
- Match the message to a client.

TcpServer (Internal)

```
// Gather outbound message (client) header information in host format.

uint32_t h_message_tag              = h_outbound_message->m_header.m_tag;
uint32_t h_message_size             = h_outbound_message->m_header.m_size;
uint32_t h_message_transaction      = h_outbound_message->m_header.m_transaction;

// Convert to network format (big-endian)

uint32_t n_message_tag              = htonl( h_message_tag );
uint32_t n_message_size             = htonl( h_message_size );
uint32_t n_message_transaction      = htonl( h_message_transaction );

// Create outbound client header in network format

n_outbound_message_header.m_tag         = n_message_tag;
n_outbound_message_header.m_size        = n_message_size;
n_outbound_message_header.m_transaction = n_message_transaction;

// Gather outbound message data

char*    message_data               = h_outbound_message->m_data;
```

- There are standard functions for network<->host translation.
- Network byte order at the transport layer is BIG ENDIAN.
- We can also say the PPU is a NETWORK BYTE ORDER machine.
- i.e. If you assume the code will only run on a big-endian machine, you can assume it will only run with network byte order.

TcpServer (Internal)

```
    int write_header_valid = WriteSocketData( message_client_ndx,
                                             (char*)&n_outbound_message_header,
                                              sizeof(TcpClientMessageHeader) );

    if (!write_header_valid)
    {
        // Drop message.
        goto CLEANUP_OUTBOUND_MESSAGE;
    }

    int write_data_valid = WriteSocketData( message_client_ndx, message_data,
                                            h_message_size );

    if (!write_data_valid)
    {
        // Drop message.
        goto CLEANUP_OUTBOUND_MESSAGE;
    }

    CLEANUP_OUTBOUND_MESSAGE:

    m_call_free( h_outbound_message->m_data );
    m_call_free( h_outbound_message );

    outbound_message_count--;
}
```

- Push data over socket.
- If write fails, we've lost the connection – just drop the packet.
- When the client re-connects, they will need to re-sync anyway.

EchoServer

- OK. Anything more interesting than this is not going to fit on these slides.
- We probably have about 15-20 minutes (?) left – let's open up TcpServer.cpp and walk through it.
- But at this point you have the main concepts that are used.