VGP393C – Week 6

- Agenda:
  - Atomic Operations
  - Non-blocking Algorithms
  - Windows threading API, part 2

Atomic Operations

- What is an “atomic operation”?

Atomic Operations

- What is an “atomic operation”?

> A “set of operations that can be combined so that they appear to the rest of the system to be a single operation...[1]”

- What does this *mean*?

[1] http://en.wikipedia.org/wiki/Atomic_(computer_science)

Atomic Operations

- What is an "atomic operation"?

A "set of operations that can be combined so that they appear to the rest of the system to be a single operation...[1]"

- What does this *mean*?
  - An instruction that performs a read-modify-write cycle that cannot be interrupted or executed out-of-order with respect to other processors in the system
  - Think of it as a really small, hardware implemented critical section

[1] http://en.wikipedia.org/wiki/Atomic_(computer_science)

Atomic Operations

- Example: `TAS` instruction on 68000
  - Reads a byte from a memory location
  - Writes the value back with the high bit set
  - Tests the original high bit and sets the condition codes

Atomic Operations

- Example: `TAS` instruction on 68000
  - Reads a byte from a memory location
  - Writes the value back with the high bit set
  - Tests the original high bit and sets the condition codes

Performed with the bus "locked"

Atomic Operations

- Example: `TAS` instruction on 68000
  - Reads a byte from a memory location
  - Writes the value back with the high bit set
  - Tests the original high bit and sets the condition codes

Performed with the bus “locked”

- Example: `XCHG` instruction on 8086
  - Reads a byte from a memory location
  - Writes a byte from a register to the memory location
  - Stores the byte from memory in the register

Atomic Operations

- Example: `TAS` instruction on 68000
  - Reads a byte from a memory location
  - Writes the value back with the high bit set
  - Tests the original high bit and sets the condition codes
- Example: `XCHG` instruction on 8086
  - Reads a byte from a memory location
  - Writes a byte from a register to the memory location
  - Stores the byte from memory in the register

Performed with the bus “locked”

Atomic Operations

- Spin-lock using `XCHG` on x86:

```
        movl        %eax, $1
1:      lock xchg   %eax, [%ebx]
        test        %eax, %eax
        jnz         1
```

Atomic Operations

- Spin-lock using `XCHG` on x86:

```
      movl         %eax, $1
1:    lock xchg    %eax, [%ebx]
      test         %eax, %eax
      jnz          1
```

  - The `lock` prefix is added on later x86 processors and allows other instructions to be atomic

Atomic Operations

- Modern processors support a variety of atomic operations
  - Increment / decrement
  - Add / subtract
  - And, or, xor, etc.
  - Exchange
  - Compare and swap

Atomic Operations

- Compare-and-swap is extremely useful, if a bit complex:

```
bool cmpxchg(int *mem, int compare, int new_value)
{
    if (*mem == compare) {
        *mem = new_value;
        return true;
    } else {
        return false;
    }
}
```

  - We'll see how this is useful in a bit...

Atomic Operations

- Windows API provides interfaces to many of these common operations:
  - `InterlockedIncrement` – Increment a 32-bit int
  - `InterlockedDecrement` – Decrement a 32-bit int
  - `InterlockedExchangeAdd` – Add a value to a 32-bit int and store the result
  - `InterlockedCompareExchange` – Compare memory to a reference value and set memory to new value if it matches the reference
    - Also `InterlockedCompareExchangePointer` and `InterlockedCompareExchange64`

Non-blocking Algorithms

- Atomic operations can be used to implement certain algorithms *without* other synchronization

Non-blocking Algorithms

- Shared counter
  - A counter that can be incremented, decremented, and tested
    - This is how we test for completion in the Mandelbrot generator
  - The increment, decrement, and test operations could be protected using a lock
  - Or...

Non-blocking Algorithms

```
class shared_counter {
public:
    void init(int value)
    {
        count = value;
    }

    bool add(int value)
    {
        return (InterlockedExchangeAdd(& count, value) == 0);
    }

private:
    volatile int count;
};
```

Non-blocking Algorithms

- Most non-blocking algorithms look fairly similar:

```
void non_blocking_foo(volatile int *x)
{
    int old_value, new_value, ref_value;

    do {
        old_value = *x;
        new_value = do_something(old_value);
        ref_value =
            InterlockedCompareExchange(x, new_value,
                                       old_value);
    } while (ref_value != old_value);
}
```

Non-blocking Algorithms

- Non-blocking singly-linked list enqueue:

```
void list::enqueue(node *n)
{
    node *old;

    do {
        n->next = head;
        old =
            InterlockedCompareExchangePointer(&head,
                                              n,
                                              n->next);
    } while (old != n->next);
}
```

Non-blocking Algorithms

- Non-blocking singly-linked list dequeue:

```
node *list::dequeue(void)
{
    node *old, *node, *next;

    do {
        node = head;
        next = node->next;
        old =
            InterlockedCompareExchangePointer(&head,
                                              next,
                                              node);
    } while (old != next);

    return node;
}
```

Non-blocking Algorithms

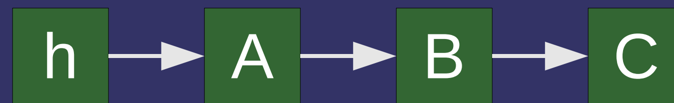
- Non-blocking singly-linked list dequeue:

```
node *list::dequeue(void)
{
    node *old, *node, *next;

    do {
        node = head;
        next = node->next;
        old =
            InterlockedCompareExchangePointer(&head,
                                              next,
                                              node);
    } while (old != next);

    return node;
}
```

WRONG!

Non-blocking Algorithms

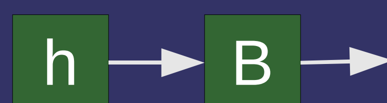
First thread:
fetch `head` → `&A`
fetch `A.next` → `&B`



Second thread:

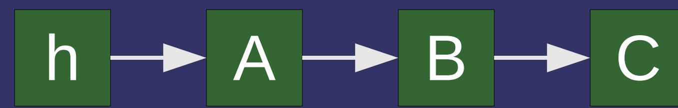
pop `A`; pop `B`; push `A`;

`cmpxchg(&head, &B &A)` → success!



Non-blocking Algorithms

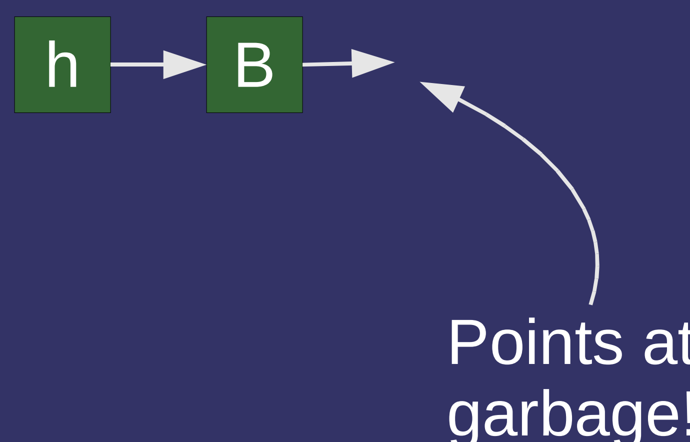
First thread:
fetch `head` → `&A`
fetch `A.next` → `&B`



Second thread:

pop `A`; pop `B`; push `A`;

`cmpxchg(&head, &B`
`&A)` → ~~success!~~
FAIL!



Non-blocking Algorithms

- For singly-linked lists, Windows provides `SLIST_HEADER`
  - `InitializeSListHead`
  - `InterlockedPushEntrySList`
  - `InterlockedPopEntrySList`
  - `InterlockedFlushSList`
  - Only available on Windows XP / Windows Server 2003 and later

Non-blocking Algorithms

- Very active area of research
  - Search for “nonblocking algorithm”
- Generally a very hard problem
  - Be wary of race conditions

Break

22-August-2008

Thread Pools

- Programs using the Fork / Join pattern often need to dynamically create and destroy lots of threads
  - High performance overhead
    - May spend more time managing threads than doing work!
  - If threads interact with the outside work (perform I/O) statically creating a few threads and a work queue may not be sufficient
  - Here a *thread pool* is the answer

Thread Pools

- A group of threads are created that feed off a work queue
  - If the queue gets too long, more threads are created
  - If the queue is empty for a long period, threads are destroyed

Thread Pools

- Several important factors in the algorithm[1]:
  - create too many threads and resources are wasted and time also wasted creating the unused threads
  - destroy too many threads and more time will be spent later creating them again
  - creating threads too slowly might result in poor client performance (long wait times)
  - destroying threads too slowly may starve other processes of resources

[1] http://en.wikipedia.org/wiki/Thread_pool_pattern

Thread Pools

- Thread pools are generally difficult to implement correctly and tune
  - Starting with Windows 2000, the system provides one for you
  - Add new tasks with:

```
BOOL QueueUserWorkItem(
    LPTHREAD_START_ROUTINE func,
    PVOID cointext,
    ULONG flags);
```

  - I/O threads should set `WT_EXECUTEINIOTHREAD` in `flags`
    - See the MSDN entry for more details

Thread Priority

- Each thread has a *priority*
  - Windows always runs “ready” threads with the highest priority first
  - High priority threads can *hog* the system and *starve* low priority threads

Thread Priority

- Set a thread's priority:

```
BOOL SetThreadPriority(
    HANDLE thread,
    int new_priority);
```

  - `new_priority` is a value between 0 and 31 or a symbolic constant:
    - `THREAD_PRIORITY_TIME_CRITICAL`
    - `THREAD_PRIORITY_HIGHEST`
    - `THREAD_PRIORITY_ABOVE_NORMAL`
    - `THREAD_PRIORITY_NORMAL`
    - `THREAD_PRIORITY_BELOW_NORMAL`
    - `THREAD_PRIORITY_LOWEST`
    - `THREAD_PRIORITY_IDLE`

Processor Affinity

- Threads are typically scheduled to run on any available processor, preferring the last processor where it was scheduled
  - Has good cache performance
  - All things being equal, this is the best choice
  - In some applications, all things are not equal
    - And by “things” we mean threads

Processor Affinity

- Consider a system with two processors, two I/O threads, and two compute threads
  - Depending on when threads are created, both compute threads may end up on the same processor
  - Since the I/O threads are often idle, this is not optimal
  - If we could tell the system to schedule an I/O thread and a compute thread on each CPU, we would win

Processor Affinity

- Two ways to modify affinity:
  - Specify the set of processors where a thread can be scheduled
  - Specify the optimal or “ideal” processor for a thread
    - On some NUMA systems, this can also set the preferred processor *node*

Processor Affinity

- Windows uses `SetThreadAffinityMask` to set the mask of processors where the thread can be scheduled:

```
DWORD_PTR SetThreadAffinityMask(
    HANDLE hThread,
    DWORD_PTR dwThreadAffinityMask);
```

Processor Affinity

- Set the ideal processor:

```
DWORD WINAPI SetThreadIdealProcessor(
    HANDLE hThread,
    DWORD dwIdealProcessor);
```

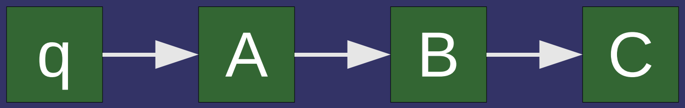
  – Windows will schedule the thread on that processor *whenever possible*
    – MSDN entry is pretty vague as to what that means

Processor Affinity

- How to use?
  - Create threads in the “idle” state
  - Set initial affinity to separate I/O and compute threads
  - Start threads running

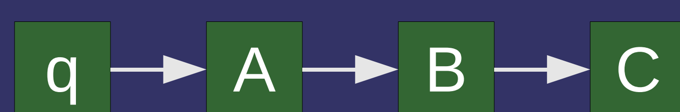
Thread-Local Storage

- Consider a fair lock implementation
  - Each waiting thread is added to a queue
  - When the lock is released, the first waiting thread wakes up
  - If a thread tries to acquire the lock and the lock is held or there are waiters, it is added to the end of the queue



Thread-Local Storage

- Fair-lock queue contains each thread *at most once*
  - Naive implementation is to allocate a node, add it to the queue
  - Nodes are released when the waiter is removed from the queue
  - This causes extra node management overhead
    - We really just want an node per thread that is persistent



Thread-Local Storage

- We want some sort of *thread-local storage*
  - Create a handle with a global ID
  - In each thread, associate some storage with that handle
    - In the fair-lock implementation, it would be the node structure
  - Code that uses the TLS obtains the per-thread storage using the handle

Thread-Local Storage

- Create a handle:

```
DWORD TlsAlloc(void);
```

- Release a handle:

```
BOOL TlsFree(DWORD dwTlsIndex);
```

- Set per-thread storage associated with handle:

```
void TlsSetValue(DWORD dwTlsIndex, void *data);
```

- Get per-thread storage associated with handle:

```
void *TlsGetValue(DWORD dwTlsIndex);
```

- See MSDN for more details
  - http://msdn.microsoft.com/en-us/library/ms686991(VS.85).aspx

Next week...

- Common multi-threading problems
  - Dead-lock / live-lock
  - Priority inversion
  - Lock contention
  - Thread-safe libraries
  - Cache abuse / memory bandwidth

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.