NOBLE Professional Edition
Application Programmers Interface (API)

May 20, 2008

Multithread and Multiprocess Components

The library is designed in an object oriented manner in order to facilitate easy adoption from most programming languages. The basic interface is provided in the C language and with a template in C++.

All components are handled as individual object classes, which all share common functionality. For each object class any number of instances can be explicitly created and freed. Of each object class there are a number of different creator functions that each uses a different implementation. Independent of which implementation was actually used to create the object instance; the operations performed on the object instance through the local handles all share the same semantics and syntax.

The user interfaces for fundamental objects that offer general services are described in the following parts: **Atomic Word Operations, Memory Manager.**

The user interfaces for the individual data structures are described in the following parts: **Stack, Queue, Deque, Priority Queue, Dictionary, List, and Snapshot.** The abstract data types each have several implementations that are named according to their characteristics. Naturally, LF stands for lock-free, WF stands for wait-free, and LB stands for lock-based. Concerning the memory requirements of the abstract data types, B stands for bounded and U stands for unbounded memory usage. Bounded means that there is an upper bound of how much memory is used for storing a certain number of items, unbounded implementations have essentially the same memory consumption related to the number of stored items but can have peaks of considerably more memory requirements due to concurrency and access patterns.

Note that although some creator functions have an argument `nrOfBlocks`, this only specifies the initial capacity of the container; as more items are stored and proceeds the initial capacity more system memory will be allocated automatically. This and other parameters normally need not to be specified, if not specified a default value will be used for respective argument and implementation.

Member functions (General)

This is a collection of member functions that apply to a majority of the object classes.

C / C++ Syntax	*Description.*
`int GetParameter(int parameter);` `void *GetParameter(int parameter);`	Gets local or global run-time

	parameters of a/an *Object* instance
`bool SetParameter(int parameter, int value);` `bool SetParameter(int parameter, void *value);`	Sets local or global run-time parameters of a/an *Object* instance.

Atomic Word Operations

Creator functions

The ***NBL::Word*** object classes can be created using three different implementations. Two implementations are lock-free and one is wait-free.

C / C++ Syntax	*Description.*
`namespace NBL {` `  template <int> class Word {`	
`static Word<int>* CreateWF_B();`	Creates a new instance using a wait-free implementation.
`static Word<int>* CreateWF_CASN(int nrOfThreads, int nrOfWords);`	Creates a new instance using a wait-free implementation.
`static Word<int>* CreateWF_LL(int nrOfVariables, int nrOfNodes);`	Creates a new instance using a wait-free implementation.

The implementations of the atomic word object support a subset of the functionality, either basic (B) or extended functionality of either multi-word updates (CASN) or load-linked updates (LL).

Member functions

C / C++ Syntax	*Description.*
`bool Init(void *word, int value);`	Initializes a memory word for operation with the word object.
`void Deinit(void *word);`	De-initializes a memory word for operation with the word object.
`int Read(void *word);`	Reads the content of a memory word.
`void Write(void *word, int value);`	Writes a value to the content of a memory word.
`int Add(void *word, int value);`	Adds a value to the content of a memory word.
`int Swap(void *word, int value);`	Exchanges the content of a memory word.
`int Op(void *word, int (*fn)(int old, int arg), int value);`	Updates the content of a memory word using a custom operation.
`bool CAS(void *word, int old, int new);`	Conditionally updates the content of a memory word.
`bool CASN(void **words, int *olds, int *news);`	Conditionally updates the contents of a collection of memory words.
`int LL(int index, void *word);`	Reads the content of a memory word,

	and starts surveillance for further updates.
`bool VL(int index, void *word);`	Checks if the content of a memory word has been updated since the start of surveillance.
`bool SC(int index, void *word, int value);`	Updates the content of a memory word only if the content has not been updated since the start of the surveillance.

Example

C++ Syntax	***Description.***
`int values[2];` `NBL::Word<int> *word;`	Globals

`word = NBL::Word<int>::CreateWF_CASN();`
`word->Init(values+0,0);`
`word->Init(values+1,0);`
`...spawn and run threads...`
`word->Deinit(values+0);`
`word->Deinit(values+1);`
`delete word;`
	Main procedure

`retry:`
`void *addrs[2];`
`int olds[2];`
`int news[2];`
`for(int i=0;i<2;i++) {`
`  addrs[i] = values+i;`
`  olds[i]=word->Read(addrs[i]);`
`  news[i]=olds[i]+1;`
`}`
`if(!word->CASN(addrs,olds,news)) goto retry;`
	Thread x

The example program above creates a thread that atomically increments two integers in a wait-free manner.

Memory Management

Creator functions

The ***NBL::Memory*** object class can be created using thirteen different implementations. Twelve implementations are lock-free and one is wait-free. Five implementations support fixed-size memory blocks, four implementations support multi-size memory blocks, and four implementations support arbitrary-size memory blocks. The ***NBL::Memory*** object is primarily used for handling custom-designed value objects to be used with various container objects, and supports safe references to dynamically allocated memory objects such that each individual thread can safely access the contents of these objects although the objects may be concurrently logically deleted and later garbage collected.

C / C++ Syntax	***Description.***
`namespace NBL {` `  template <typename T> class Memory {`	
`static Memory<T>* CreateLF_SLB(int nrOfBlocks, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_SUU(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF MLB(int nrOfBlocks, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_WLB(int nrOfBlocks, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateWF_SUU(int nrOfThreads, int nrOfBlocks);`	Creates a new instance using a wait-free implementation.
`static Memory<T>* CreateLF_CSLB( NBLMemorySizeClass *sizeClasses, int nrSizeClasses, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_CSUU( NBLMemorySizeClass * sizeClasses, int nrSizeClasses);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_CMLB( NBLMemorySizeClass * sizeClasses, int nrSizeClasses, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_CWLB( NBLMemorySizeClass * sizeClasses, int nrSizeClasses, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_HSLB(int heapSize, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_HSUU(int heapSize);`	Creates a new instance using a lock-free implementation.
`static Memory<T>* CreateLF_HMLB(int heapSize, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.

`static Memory<T>* CreateLF HWLB(int heapSize, int nrLocalRefs);`	Creates a new instance using a lock-free implementation.

The memory manager implementations either support fixed-size, a selection of sizes (C), or arbitrary size (H) of memory blocks for allocation. The memory reclamation mechanism can either be strong (S), medium (M), or weak (W) dependent on the ability to keep watch over both global and local pointers. The number of local pointers that can be handled by each thread can either be limited (L) or unlimited (U).

Member functions

C / C++ Syntax	*Description*
`T *AllocBlock();`	Allocates a new memory block of fixed size.
`T *AllocClass(int sizeClass);`	Allocates a new memory block of selected size-class.
`T *AllocSize(int size);`	Allocates a new memory block of arbitrary size.
`void DeleteBlock(T *block);`	Frees a memory block when possible.
`T *DeRefLink(T **link);`	Dereferences a shared memory pointer.
`void CopyRef(T *block);`	Copies a safe reference.
`void ReleaseRef(T *&block);`	Releases a safe reference.
`void StoreRef(T **link, T *block);`	Stores a reference in a shared memory pointer.
`bool CASRef(T **link, T *old, T *_new);`	Atomically updates a reference in a shared memory pointer.

Example

C++ Syntax	*Description.*

`class MyObject {`
`public:`
`  int x;`
`  int y;`
`  int z;`
`};`
`NBL::Memory<MyObject> * memory;`
`MyObject *myObj = NULL;`
	Globals

`memory =`
`NBL::Memory<MyObject>::CreateLF_SUU();`
`...spawn and run threads...`
`delete memory;`
	Main procedure

`MyObject * obj1 = memory->AllocBlock();`	Thread x

```
obj1->x = 1;
obj1->y = 2;
obj1->z = 3;
memory->StoreRef(&myObj,obj1);
memory->ReleaseRef(obj1);
```

```
MyObject * obj1 = memory->DeRefLink(&myObj);
if(obj1!=NULL) {
    ...
    memory->StoreRef(&myObj,NULL);
    memory->ReleaseRef(obj1);
}
```

Thread y

The example program above creates two threads that pass a custom data object between each other. The data object is allocated and reclaimed by a lock-free memory manager.

Shared Stack

Creator functions

The ***NBL::Stack*** object classes can be created using three different implementations. Two implementations are lock-free and one is wait-free.

C / C++ Syntax	*Description.*
`namespace NBL {` `  template <typename T> class Stack {`	
`static Stack<T>* CreateLF_B(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Stack<T>* CreateLF_U(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Stack<T>* CreateLB();`	Creates a new instance using a lock-based implementation.

Member functions

C / C++ Syntax	*Description*
`bool Push(T *item);`	Pushes a new item on the stack.
`T *Pop();`	Pops an item from the stack.

Example

C++ Syntax	*Description.*
`int values[2]={1,2};` `NBL::Stack<int> *stack;`	Globals

`stack = NBL::Stack<int>::CreateLF_B();`
`...spawn and run threads...`
`delete stack;`
	Main procedure

`stack->Push(values+0);`	Thread x

`int *value = stack->Pop();`
`if(value!=NULL) {`
`  ...`
`}`
	Thread y

The example program above creates two threads that pass a reference to an integer between each other; from thread x to thread y. The means of data transfer is a stack with lock-free operations.

Shared Queue

Creator functions

The ***NBL::Queue*** object class can be created using five different implementations. Three implementations are lock-free, one is wait-free, and one is lock-based.

C / C++ Syntax	***Description.***
`namespace NBL {` `  template <typename T> class Queue {`	
`static Queue <T>* CreateLF DB(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Queue <T>* CreateLF_DU(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Queue<T>* CreateLF_SB(int nrNodes);`	Creates a new instance using a lock-free implementation.
`static Queue<T>* CreateWF_SS(int nrNodes);`	Creates a new instance using a wait-free implementation.
`static Queue <T>* CreateLB();`	Creates a new instance using a lock-based implementation.

The Queue can be based on either a static (S) or dynamic (D) underlying data structure. Some implementations only support limited concurrency, e.g. single reader and single writer (SS).

Member functions

C / C++ Syntax	***Description***
`bool Enqueue(T *item);`	Puts a new item on top of the queue.
`T *Dequeue();`	Removes an item from bottom of the queue.
`int Size();`	Estimates the current number of items in the queue.
`bool IsEmpty();`	Answers whether the queue is empty or not.

Example

C++ Syntax	***Description.***
`int values[2]={1,2};` `NBL::Queue<int> *queue;`	Globals
`queue = NBL::Queue<int>::CreateLF_DB();`	Main procedure

...spawn and run threads... delete queue;	
queue->Enqueue(values+0);	Thread x

int *value = queue->Dequeue();
if(value!=NULL) {
...
}
	Thread y

The example program above creates two threads that pass a reference to an integer between each other; from thread x to thread y. The means of data transfer is a queue with lock-free operations.

Shared Deque

Creator functions

The ***NBL::Deque*** object class can be created using four different implementations. Three implementations are lock-free and one is lock-based.

C / C++ Syntax	*Description.*
`namespace NBL {` `  template <typename T> class Deque {`	
`static Deque<T>* CreateLF_HB(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Deque<T>* CreateLF_HU(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Deque<T>* CreateLF_LB(int nrOfBlocks, int nrOfThreads);`	Creates a new instance using a lock-free implementation.
`static Deque<T>* CreateLB();`	Creates a new instance using a lock-based implementation.

The Deque can either offer high (H) or limited (L) level of parallelism.

Member functions

C / C++ Syntax	*Description*
`bool PushLeft(T *item);`	Puts a new item on top of the deque.
`bool PushRight(T *item);`	Puts a new item on bottom of the deque.
`T *PopLeft();`	Removes an item from top of the deque.
`T *PopRight();`	Removes an item from bottom of the deque.

Example

C++ Syntax	*Description.*
`int values[2]={1,2};` `NBL::Deque<int> *deque;`	Globals

`deque = NBL::Deque<int>::CreateLF_HB();`
`...spawn and run threads...`
`delete deque;`
	Main procedure

`deque->PushLeft(values+0);`	Thread x

```
deque->PushRight(values+1);
```

```
int *value = deque->PopRight();
if(value!=NULL) {
    ...
}
```

Thread y

```
int *value = deque->PopLeft();
if(value!=NULL) {
    ...
}
```

Thread z

The example program above creates three threads that pass references to two integers between each other; from thread x to thread y and from thread x to thread z. The means of data transfer is a deque with lock-free operations.

Shared Priority Queue

Creator functions

The ***NBL::PQueue*** object class can be created using three different implementations. Two implementations are lock-free and three is lock-based. Three implementations offer expected logarithmic search times and two implementations offer logarithmic search time. All implementations support custom-designed functions for priority comparison and custom data type of priorities.

C / C++ Syntax	***Description.***

`namespace NBL {`
`  template <typename T, typename P = int>`
`  class PQueue {`	

`static PQueue<T,P>* CreateLF_EB(int nrOfBlocks, int avgNodes);`	Creates a new instance using a lock-free implementation.
`static PQueue<T,P>* CreateLF_EU(int nrOfBlocks, int avgNodes);`	Creates a new instance using a lock-free implementation.
`static PQueue<T,P>* CreateLB_E(int avgNodes);`	Creates a new instance using a lock-based implementation.
`static PQueue<T,P>* CreateLB_SD(int nrOfBlocks, int nrOfThreads);`	Creates a new instance using a lock-based implementation.
`static PQueue<T,P>* CreateLB_DD();`	Creates a new instance using a lock-based implementation.

The Priority Queue can offer an expected logarithmic (E), deterministic logarithmic (D), or linear (L) time complexity for searches with respect to their size.

Member functions

C / C++ Syntax	***Description***
`bool Insert(int priority, T *item);` `bool Insert(P *priority, T *item);`	Inserts a new item.

`T *DeleteMin();`
`T *DeleteMin(int *priority);`
`T *DeleteMin(P **priority);`
	Removes the item with the lowest priority.

`T *FindMin();`
`T *FindMin(int *priority);`
`T *FindMin(P **priority);`
	Finds the item with the lowest priority.

Example

C++ Syntax	***Description.***
`int values[2]={1,2};`	Globals

`NBL::PQueue<int,int> *pqueue;`	

`pqueue = NBL::PQueue<int,int>::CreateLF_EB();`
`...spawn and run threads...`
`delete pqueue;`
	Main procedure

`pqueue->Insert(values+0,1);`	Thread x
`pqueue->Insert(values+1,2);`	Thread y

`int *value = pqueue->DeleteMin();`
`if(value!=NULL) {`
`  ...`
`}`
	Thread z

The example program above creates three threads that pass references to two integers between each other; from thread x to thread z or from thread y to thread z. The means of data transfer is a priority queue with lock-free operations.

Shared Dictionary

Creator functions

The ***NBL::Dictionary*** object class can be created using three different implementations. Three implementations are lock-free and one is lock-based. Three implementations offer expected logarithmic search times and one implementation offer linear search time. All implementations support custom-designed functions for key comparison and custom data type of keys.

C / C++ Syntax	***Description.***

`namespace NBL {`
`  template <typename T, typename K = int>`
`  class Dictionary {`	

`static Dictionary<T,K>* CreateLF_EB(int nrOfBlocks, int avgNodes);`	Creates a new instance using a lock-free implementation.
`static Dictionary<T,K>* CreateLF_EU(int nrOfBlocks, int avgNodes);`	Creates a new instance using a lock-free implementation.
`static Dictionary<T,K>* CreateLF_LB(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static Dictionary<T,K>* CreateLB_E(int avgNodes);`	Creates a new instance using a lock-based implementation.

The Dictionary can offer an expected logarithmic (E), deterministic logarithmic (D), or linear (L) time complexity for searches with respect to their size.

Member functions

C / C++ Syntax	***Description***
`bool Insert(int key, T *item);` `bool Insert(K * key, T *item);`	Inserts a new association.

`bool Update(int key, T *item);`
`bool Update(K * key, T *item);`
`bool Update(int key, T *item, T **old);`
`bool Update(K * key, T *item, T **old);`
	Updates an existing association.

`T *Delete(int key);` `T *Delete(K * key);`	Deletes an association.
`T *Find(int key);` `T *Find(K * key);`	Finds the value associated with a certain key.

Example

C++ Syntax	***Description.***
`class MyObject {`	Globals

`public:`
`    int x;`
`    int y;`
`    int z;`
`};`
`MyObject values[2]={{1,2,3},{4,5,6}};`
`NBL::Dictionary<MyObject,int> *dictionary;`	

`dictionary = NBL::Dictionary<MyObject`
`,int>::CreateLF_EB();`
`...spawn and run threads...`
`delete dictionary;`
	Main procedure

`dictionary->Insert(values+0,1);`	Thread x
`dictionary->Insert(values+1,2);`	Thread y

`MyObject *value = dictionary->Find(1);`
`if(value!=NULL) {`
`   ...`
`}`
	Thread z

The example program above creates three threads that share references to two custom data objects between each other, where each data object is associated with a given integer. The means of data sharing is a dictionary with lock-free operations.

Shared List

Creator functions

The ***NBL::List*** object class can be created using five different implementations. Three implementations are lock-free and two is lock-based. Two of the implementations are singly linked and three are doubly linked. All implementations support continued traversals from positions with elements possibly deleted by concurrent operations.

C / C++ Syntax	***Description.***
`namespace NBL { template <typename T> class List {`	
`static List<T> *CreateLF_SU(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static List<T> *CreateLB_S();`	Creates a new instance using a lock-free implementation.
`static List<T> *CreateLF_DB(int nrOfBlocks);`	Creates a new instance using a lock-free implementation.
`static List<T> *CreateLF_DU(int nrOfBlocks);`	Creates a new instance using a lock-based implementation.
`static List<T> *CreateLB_D();`	

The List can be either singly (S) or doubly (D) linked.

Member functions

C / C++ Syntax	***Description***
`bool InsertBefore(T *item);`	Inserts a new element directly before the current position.
`bool InsertAfter(T *item);`	Inserts a new element directly after the current position.
`T *Delete();`	Deletes the element at the current position.
`T *Read();`	Reads the element at the current position.
`void First();`	Sets the cursor position to point directly before the first element.
`void Last();`	Sets the cursor position to point directly after the last element.
`bool Next();`	Traverses the cursor position one step forwards.

`bool Previous();`	Traverses the cursor position one step backwards.

Example

C++ Syntax	***Description.***
`int values[2]={1,2};` `NBL::List<int> *list;`	Globals

`list = NBL::List<int>::CreateLF_DB();`
`...spawn and run threads...`
`delete list;`
	Main procedure

`list->First();` `list->InsertAfter(values+0);`	Thread x
`list->Last();` `list->InsertBefore(values+1);`	Thread y

`list->First();`
`while(list->Next()) {`
`  int *value = list->Read();`
`  if(value!=NULL) {`
`  ...`
`  }`
`}`
	Thread z

`list->Last();`
`while(list->Previous()) {`
`  int *value = list->Delete();`
`  if(value!=NULL) {`
`  ...`
`  }`
`}`
	Thread w

The example program above creates four threads that share references to two integers, where each data object is associated with a given integer. The means of data sharing is a doubly-linked list with lock-free operations.

Shared Snapshot

Creator functions

The ***NBL::Snapshot*** object class can be created using three different implementations. Three implementations are wait-free and one is lock-based.

C / C++ Syntax	***Description.***
`namespace NBL {` `  template <typename T> class Snapshot {`	
`static Snapshot<T>* CreateWF_SS(int components);`	Creates a new instance using a wait-free implementation.
`static Snapshot<T>* CreateWF_SM(int components, int writers);`	Creates a new instance using a wait-free implementation.
`static Snapshot<T>* CreateWFR_SM(int components, int *cycles);`	Creates a new instance using a wait-free implementation.
`static Snapshot<T>* CreateLB(int components);`	Creates a new instance using a lock-based implementation.

The Snapshot can support a single (S) scanner together with either single (S) or multiple (M) updaters.

Member functions

C / C++ Syntax	***Description***
`void Scan(T *values[]);`	Scans the components of the snapshot object.
`void Update(int component, T *value);`	Updates a single component of the snapshot object.

Example

C++ Syntax	***Description.***
`int values[2]={1,2};` `NBL::Snapshot<int> *snapshot;`	Globals

`snapshot =`
`NBL::Snapshot<int>::CreateWF_SM(2);`
`...spawn and run threads...`
`delete snapshot;`
	Main procedure

`snapshot->Update(0,values+0);`	Thread x
`snapshot->Update(1,values+1);`	Thread y
`int scan[2];` `snapshot->Scan(scan);`	Thread z

Parallel Scalable Solutions AB, Box 916, SE-501 10 BORÅS, SWEDEN.
info@pss-ab.com , www.pss-ab.com