NOBLE
Professional Edition
v2.2

DEVELOPERS MANUAL

March 3, 2009

Preface

NOBLE Professional Edition is a software library for multi-thread and multi-process environments. It is developed by Dr. Håkan Sundell with overall directions by Prof. Dr. Philippas Tsigas. NOBLE Professional Edition is a new product developed from scratch, influenced by the NOBLE research project. NOBLE Professional Edition is developed with a much larger scope than the research project, and does not include any remains from it besides the name, concept, and approach. The NOBLE research project was initiated in 2001 by Prof. Dr. Philippas Tsigas at the Department of Computer Science at Chalmers University of Technology in Sweden, with development work mainly conducted by Dr. Håkan Sundell.

Contents

Chapter 1

Introduction

Noble is a software package. Main areas of applications are within the Parallel and Real-Time Systems community where concurrency is needed.

Noble consists of several implementations of commonly used data structures. This kind of structures like stacks, queues, lists etceteras need some kind of synchronization when used in concurrent environments. These environments can be multitasking systems, parallel computers as well as distributed real-time systems. The most common synchronization method involves some kind of locks, which enforces blocking.

Noble provides several non-blocking implementations of common data structures as well as lock based ones. The users can change synchronization method transparently to the one that suits best for the current application, even in run-time. With Noble the users can take benefits of non-blocking protocols without thorough knowledge in the wait and lock-free community.

Noble is easy to use and supports most programming languages as it is written in C. Several platforms and computer architectures are supported and more are planned in the future.

1.1 Getting Started

1.1.1 C

Use a text editor to create a file program.c, with the following code:

```
#include <stdio.h>

#include "Noble.h"

int main(int argc, char **argv)
```

```
{
  NBLStackRoot *stack; /* Stack object */
  NBLStack *handle; /* Stack handle */
  int item=10;
  int *data; /* Pointer to the data */

  /* Initialize the stack object */
  stack = NBLStackCreateLF_B(100);

  /* Create a stack handle */
  handle = NBLStackGetHandle(stack);

  /* Push the data on the stack */
  NBLStackPush(handle, &item);

  /* Pop the data off the stack */
  data = NBLStackPop(handle);

  /* Print the data */
  printf("%d\n",(int)*data);

  /* Free the handle */
  NBLStackFreeHandle(handle);

  /* Free the stack */
  NBLStackFree(stack);

  return 0;
}
```

Put the following Makefile in the same directory. It uses the gnu c compiler. You may have to change the path to the include files.

```
# Makefile for the stack program
CC = gcc

SRC = program.c

OBJS = program.o

CFLAGS = -I../Include/

all:    test_noble

test_noble:     $(OBJS)
        $(CC) $(OBJS) -L/usr/lib ../Lib/libNOBLE.a

program.o:      program.c

.c.o:
        $(CC) $(CFLAGS) -c $*.c -o $*.o
```

Use the make command to compile the file.

1.1.2 C++

Use a text editor to create a file program.cpp, with the following code:

```
#include <stdio.h>

#include "NobleCPP.h"

int main(int argc, char **argv)
{
  NBL::Stack<int> *stack; /* Stack object */
  int item=10;
  int *data; /* Pointer to the data */

  /* Initialize the stack object */
  stack = NBL::Stack<int>::CreateLF_U();

  /* Push the data on the stack */
  stack->Push(&item);

  /* Pop the data off the stack */
  data = stack->Pop();

  /* Print the data */
  printf("%d\n",(int)*data);

  /* Free the stack */
  delete stack;

  return 0;
}
```

1.2 How to read the manual pages

The descriptions of the functions have the following parts:

Name The name of the function.

Description Description of what the function does

Synopsis The definition of the function, and its operations. Similar to the include file for the data structures.

Parameters Description of the individual parameters that should be provided when calling the function.

Return Values Description of the different values that this function can produce as a result of execution.

Remarks Extra information about this function that has to be understood and properly handled before using this function.

Requirements What kind of system requirements that is demanded to be able to use this function.

See Also A list of similar or closely connected functions.

1.3 Technical Information

This section describes what files are included in the library, and the systems it is implemented on.

1.3.1 Source Code Package

The following files and subdirectories are included:

Include/		Include directory.
	Noble.h	The main C/C++ declarations of Noble
Lib/		Library files.
	Noble.lib	The library file to link with the executable (Win32).
	libNoble.a	The library file to link with the executable (Unix).

1.3.2 Usage

To use a specific data structure the header file for Noble has to be included into the program.

#include ”Noble.h”

Chapter 2

Components

The library is designed to provide easy access to the components, with names that describe what the operations do and follow common practice. Since the data structures are very common, most of the names are obvious. All functions have the prefix NBL to avoid name conflicts in the development environment.

The interface has been specified to make it easy to change from one implementation of a data structure to another. For example, this makes it easy to change from a lock-based to a lock-free implementation of a particular data structure. The interface is designed with an object-oriented approach. In order to facilitate dynamic attachment of threads vs. processes to the shared objects, the objects are accessed via a two-level hierarchy. The hierarchy of the shared object in NOBLE constitutes of one *root* object and an arbitrary number of *handle* objects. The root object is created as an instance of shared data object (e.g. when NBL*Object*CREATE*X* is called for the specific object and implementation). Each process vs. thread can attach to a root object and need to create a corresponding handle in order to perform operations on the shared data object instance. 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 See Chapter 3.

Memory Manager See Chapter 4.

The user interfaces for the individual data structures are described in the following parts:

Stack See Chapter 5.

Queue See Chapter 6.

Deque See Chapter 7.

Priority Queue See Chapter 8.

Dictionary See Chapter 9.

List See Chapter 10.

Snapshot See Chapter 11.

2.1 Availability

The Demo version of NOBLE, which can freely be used for evaluation purposes only, does not permit access to all implementations available in the fully licensed version. Below follows a description of the available implementations for respective data structures and distribution package:

		Demo version	Professional Edition
Stack	**LF_B**	-	X
	LF_U	X	X
	LB	X	X
Queue	**WF_SS**	-	X
	LF_DB	-	X
	LF_DU	X	X
	LF_SB	-	X
	LB	X	X
Deque	**LF_HB**	-	X
	LF_HU	X	X
	LF_LB	-	X
	LB	X	X
PQueue	**LF_EB**	-	X
	LF_EU	X	X
	LB_SD	X	X
	LB_DD	X	X
	LB_E	X	X
Dictionary	**LF_EB**	-	X
	LF_EU	X	X
	LF_LB	-	X
	LB_E	X	X
List	**LF_SU**	X	X
	LF_DB	-	X
	LF_DU	X	X
	LB_S	X	X
	LB_D	X	X

		Demo version	Professional Edition
Snapshot	**WF_SS**	-	X
	WF_SM	-	X
	WFR_SM	-	X
	LB	X	X
Word	**WF_B**	X	X
	WF_CASN	-	X
	LF_LL	-	X
Memory	**LF_SLB**	-	X
	LF_SUU	X	X
	LF_MLB	-	X
	LF_WLB	-	X
	LF_CSLB	-	X
	LF_CSUU	X	X
	LF_CMLB	-	X
	LF_CWLB	-	X
	LF_HSLB	-	X
	LF_HSUU	X	X
	LF_HMLB	-	X
	LF_HWLB	-	X
	WF_SUU	-	X

Attempted usage of an unavailable implementation will result in the automatic use of the nearest (compatible) available implementation, i.e., the standard lock-based implementation.

2.2 General Functions

This is a collection of functions that apply to a majority of the shared data structures.

2.2.1 C

NBL*Object*Free Frees the memory used by a/an *Object* instance. See Section 2.2.3.

NBL*Object*GetHandle Creates a new local handle to a/an *Object* instance. See Section 2.2.4.

NBL*Object*FreeHandle Frees the memory used by a handle of a/an *Object* instance. See Section 2.2.5.

NBL*Object*GetParameter Gets local or global run-time parameters of a/an *Object* instance. See Section 2.2.6.

NBL*Object*SetParameter Sets local or global run-time parameters of a/an *Object* instance. See Section 2.2.7.

In the previous functions, *Object* denotes any of Memory, Stack, Queue, Deque, PQueue, Dictionary, List or Snapshot.

2.2.2 C++

NBL::*Object*::GetRootParameter Gets local or global run-time parameters of a/an *Object* instance. See Section 2.2.6.

NBL::*Object*::SetRootParameter Sets local or global run-time parameters of a/an *Object* instance. See Section 2.2.7.

NBL::*Object*::GetLocalParameter Gets local or global run-time parameters of a/an *Object* instance. See Section 2.2.6.

NBL::*Object*::SetLocalParameter Sets local or global run-time parameters of a/an *Object* instance. See Section 2.2.7.

2.2.3 NBL*Object*Free

Frees all the global shared memory used by a/an *Object* instance.

Syntax C

```
void NBLObjectFree(
   NBLObjectRoot* object
);
```

Syntax C++

```
delete object;
```

Parameters

object
: [in] A pointer to a/an *Object* instance.

Return Values

This function does not return any value.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

2.2.4 NBL*Object*GetHandle

Creates a new local handle to a/an *Object* instance.

Syntax C

```
NBLObject * NBLObjectGetHandle(
  NBLObjectRoot* object
);
```

Parameters

object
[in] A pointer to a/an *Object* instance.

Return Values

If successful the function returns a pointer to a local handle of a/an *Object* instance. Otherwise it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

2.2.5 NBL*Object*FreeHandle

Frees all local memory used by a handle of a/an *Object* instance.

Syntax C

```
void NBLObjectFreeHandle(
  NBLObject* handle
);
```

Parameters

handle
[in] A pointer to local handle of a/an *Object* instance.

Return Values

This function does not return any value.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

2.2.6 NBL*Object*GetParameter

Gets local or global run-time parameters of a/an *Object* instance.

Syntax C

```
void * NBLObjectGetParameter(
  NBLObject* handle,
  NOBLE_PARAM param
);
```

```
void * NBLObjectGetParameter(
  NBLObjectRoot* object,
  NOBLE_PARAM param
);
```

Syntax C++

```
void * NBL::Object::GetLocalParameter(
  NOBLE_PARAM param
);
```

```
void * NBL::Object::GetRootParameter(
  NOBLE_PARAM param
);
```

Parameters

object
: [in] A pointer to a/an *Object* instance.

handle
: [in] A pointer to a local handle of a/an *Object* instance.

param
: [in] The local or global run-time parameter to get the current value for.

Return Values

If successful the function returns the value associated with the given parameter. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

2.2.7 NBL*Object*SetParameter

Sets local or global run-time parameters of a/an *Object* instance.

Syntax C

```
int NBLObjectSetParameter(
  NBLObject* handle,
  NOBLE_PARAM param,
  void *value
);
```

```
int NBLObjectSetParameter(
  NBLObjectRoot* object,
  NOBLE_PARAM param,
  void *value
);
```

Syntax C++

```
bool NBLObjectSetLocalParameter(
  NOBLE_PARAM param,
  void *value
);
```

```
bool NBLObjectSetRootParameter(
  NOBLE_PARAM param,
  void *value
);
```

Parameters

object
[in] A pointer to a/an *Object* instance.

handle
[in] A pointer to a local handle of a/an *Object* instance.

param
[in] The local or global run-time parameter to set.

value
[in] The value that the local or global run-time parameter should be set to.

Return Values

If successful the function returns true. Otherwise the parameter could not be set and the function returns false.

Remarks

On-line parameters that are set on global (i.e. object) level are inherited on local (i.e. handle) level when the respective handles are created.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

2.3 Multi-Process versus Multi-Thread

The library is by default configured for multi-thread usage, i.e. an environment with shared memory where also the code area for the library is located at the same memory address for all the threads. It is possible to configure the library to also work in a multi-process environment, i.e. an environment with shared memory where the code area for the library could be located at different memory addresses for each process. The following function is defined for activating multi-process support.

NBLObjectMultiProcessEnable Enable the multi-process usage of a particular object instance. See Section 2.3.1.

2.3.1 NBLObjectMultiProcessEnable

Enable the multi-process usage of a particular object instance.

Syntax

```
void NBLObjectMultiProcessFree(
  NBLObjectRoot* object
);
```

Parameters

object
: [in] A pointer to a/an *Object* instance, where *Object* denotes any of Memory, Stack, Queue, Deque, PQueue, Dictionary, List or Snapshot.

Return Values

This function does not return any value.

Remarks

This function must be called before the pointer to the object instance can be shared with any other process in the multi-process environment. Only after this function is called can local handles to the object instance be safely created by the other processes.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

2.4 On-Line Parameters

All objects and handles support the NBL*Object*GetParameter() and NBL*Object*GetParameter() functions, that can retrieve or change specific run-time parameters. The handles normally inherit all the parameter settings of the global objects at the time of creation of the handle.

Which certain parameters that can be tuned, are specific for each implementation of a shared data structure, and are listed together with the description of the respective object creation function.

2.4.1 Memory Parameters

Here is a list of parameters that access settings that are specific to the memory managers.

PARAM_MEM_OVERHEAD_SIZE Specifies in bytes the overhead that the memory manager imposes on each memory block. Read-only parameter.

PARAM_MEM_FIXED_ALLOC Specifies whether (true or false) the memory handler should call the default global memory handler for reserving more memory whenever the memory reserved at creation is all allocated. The default setting for this parameter is true.

PARAM_MEM_CB_RELEASE_REFS Specifies the callback function that should be called whenever the memory manager is about to free a memory block.

PARAM_MEM_CB_REDUCE_CHAIN Specifies the callback function that should be called whenever the memory manager wants to make sure that a memory block that is about to be freed does not contain any links to other memory blocks that is about to be freed.

PARAM_MEM_CB_ARG1 Specifies the first argument that should be given to the callback functions when they are called by the memory manager.

2.4.2 User-Object Memory-Manager Parameters

Here is a list of parameters that access settings that are specific for specifying what memory manager that should be used for value parameters and return values for various data structures.

PARAM_VALUE_MEMORY_HANDLE Specifies the local handle to the memory manager object that should be used for handling value parameters and return results for various data structures. Parameter is given as a pointer to `NBLMemory`. Write-only, local-only parameter.

PARAM_VALUE_MEMORY_ROOT Specifies the instance of a memory manager object that should be used for handling value parameters and return results for various data structures. Parameter is given as a pointer to `NBLMemoryRoot`. Write-only, global-only parameter.

Note that for proper usage, the previous on-line parameters have to be set on both global and local level , i.e. the PARAM_VALUE_MEMORY_ROOT parameter must be set on global level of the data structure object and the PARAM_VALUE_MEMORY_HANDLE parameter must be set on all handles to the data structure object.

C++

By using the class template for the Dictionary creation and specifying the appropriate value type, the PARAM_VALUE_MEMORY_ROOT and PARAM_VALUE_MEMORY_HANDLE parameters can be omitted by instead calling the method SetValueMemoryHandler, see Section 9.4.5.

2.4.3 Back-Off Parameters

Here is a list of parameters that affect the back-off strategy used by some implementation. The back-off is used for tuning the specific algorithm to the current situtation of contention on the shared memory, and thus achieving maximum overall performance.

PARAM_BACK_OFF_TYPE Specifies the type of back-off strategy that should be used. The different types available are:

- **BOT_LINEAR** The amount of iterations during each back-off, is increased linearly for every subsequent back-off during one operation invocation.
- **BOT_EXPONENTIAL** The amount of iterations during each back-off, is increased exponentially for every subsequent back-off during one operation invocation.

PARAM_BACK_OFF_INIT The initial amount of iterations (of local computations) that is done during each back-off instance.

PARAM_BACK_OFF_MAX The maximum amount of iterations that can be done during each back-off instance.

2.4.4 Priority Queue Parameters

Here is a list of parameters that access settings that are specific to the the priority queue data structure.

PARAM_PRI_COMPARE_FN Specifies a user-defined function that compares two priorities and decides whether pri1 $<$ pri2, pri1 $=$ pri2 or pri1 $>$ pri2, returning -1, 0 or 1 respectively. The syntax of the function is
`int comparefn(const void *pri1, const void *pri2).`

C++

By using the class template for the PQueue creation and specifying the appropriate value type, the PARAM_KEY_COMPARE_FN parameter can be omitted by instead properly defining the corresponding operators for comparison of the specified value type K:

```
bool operator < (const K &t1, const K &t2);
bool operator == (const K &t1, const K &t2);
```

2.4.5 Dictionary Parameters

Here is a list of parameters that access settings that are specific to the the dictionary data structure.

PARAM_KEY_COMPARE_FN Specifies a user-defined function that compares two keys and decides whether key1 $<$ key2, key1 $=$ key2 or key1 $>$ key2, returning -1, 0 or 1 respectively. The syntax of the function is
`int comparefn(const void *key1, const void *key2).`

C++

By using the class template for the Dictionary creation and specifying the appropriate value type, the PARAM_KEY_COMPARE_FN parameter can be omitted by instead properly defining the corresponding operators for comparison of the specified value type K:

```
bool operator < (const K &t1, const K &t2);
bool operator == (const K &t1, const K &t2);
```

Chapter 3

Atomic Word Operations

The Word object allows atomic operations on a single or a collection of memory words, i.e. the object allows transactions on a set of memory words.

The implementations available of the Word object are:

- A Wait-Free implementation offering basic (i.e. native) operations support.
- A Wait-Free implementation offering basic and transaction operations support.
- A Lock-Free implementation offering basic and surveillance operations support.

Not all of the implementations support all operations. However, all of the available implementations of the word object offer the same high-level semantics where the specific operation is supported.

The detailed semantics of these operations are described in the respective detailed description of each individual operation.

3.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

3.1.1 C

NBLWordCreateWF_B Creates a new instance of a word object using a wait-free implementation. See Section 3.3.1.

NBLWordCreateWF_CASN Creates a new instance of a word object using a wait-free implementation. See Section 3.3.2.

NBLWordCreateLF_LL Creates a new instance of a word object using a lock-free implementation. See Section 3.3.3.

NBLWordInit Initializes a memory word for use with the word object. See Section 3.4.1.

NBLWordDeinit De-initializes a memory word after use with the word object. See Section 3.4.2.

NBLWordRead Reads the content of a memory word. See Section 3.4.3.

NBLWordWrite Writes a value to the content of a memory word. See Section 3.4.4.

NBLWordAdd Adds a value to the content of a memory word. See Section 3.4.5.

NBLWordSwap Exchanges the content of a memory word. See Section 3.4.6.

NBLWordOp Updates the content of a memory word using a custom operation. See Section 3.4.7.

NBLWordCAS Conditionally updates the content of a memory word. See Section 3.4.8.

NBLWordCASN Conditionally updates the contents of a collection of memory words. See Section 3.4.9.

NBLWordLL Reads the content of a memory word, and starts surveillance for further updates. See Section 3.4.10.

NBLWordVL Checks if the content of a memory word has been updated since the start of surveillance. See Section 3.4.11.

NBLWordSC Updates the content of a memory word only if the content has not been updated since the start of the surveillance. See Section 3.4.12.

3.1.2 C++

```
namespace NBL {
  template <int> class Word {
    // Constructors
    static Word<int>* CreateWF_B();
    static Word<int>* CreateWF_CASN(int nrOfThreads, int
        nrOfWords);
    static Word<int>* CreateWF_LL(int nrOfVariables, int
        nrOfNodes);
    // Operations
```

```
    bool Init(void *address, int value);
    void Deinit(void *address);
    int Read(void *address);
    void Write(void *address, int value);
    int Add(void *address, int value);
    int Swap(void *address, int next);
    int Op(void *address, int (*fn)(int old, int arg), int
        value);
    bool CAS(void *address, int old, int next);
    bool CASN(int n, void **addresses, int *olds, int *nexts);
    int LL(int nr, void *address);
    bool VL(int nr, void *address);
    bool SC(int nr, void *address, int value);
  };
};
```

NBL::Word::CreateWF_B Creates a new instance of a word object using a wait-free implementation. See Section 3.3.1.

NBL::Word::CreateWF_CASN Creates a new instance of a word object using a wait-free implementation. See Section 3.3.2.

NBL::Word::CreateLF_LL Creates a new instance of a word object using a lock-free implementation. See Section 3.3.3.

NBL::Word::Init Initializes a memory word for use with the word object. See Section 3.4.1.

NBL::Word::Deinit De-initializes a memory word after use with the word object. See Section 3.4.2.

NBL::Word::Read Reads the content of a memory word. See Section 3.4.3.

NBL::Word::Write Writes a value to the content of a memory word. See Section 3.4.4.

NBL::Word::Add Adds a value to the content of a memory word. See Section 3.4.5.

NBL::Word::Swap Exchanges the content of a memory word. See Section 3.4.6.

NBL::Word::Op Updates the content of a memory word using a custom operation. See Section 3.4.7.

NBL::Word::CAS Conditionally updates the content of a memory word. See Section 3.4.8.

NBL::Word::CASN Conditionally updates the contents of a collection of memory words. See Section 3.4.9.

NBL::Word::LL Reads the content of a memory word, and starts surveillance for further updates. See Section 3.4.10.

NBL::Word::VL Checks if the content of a memory word has been updated since the start of surveillance. See Section 3.4.11.

NBL::Word::SC Updates the content of a memory word only if the content has not been updated since the start of the surveillance. See Section 3.4.12.

3.1.3 Limitations in Functionality

All implementations do not support all operations. The following table illustrates which operations that are supported by each implementation:

	Read	*Write*	*Add*	*Swap*	*Op*	*CAS*	*CASN*	*LL*	*VL*	*SC*
WF_B	WF	WF	WF	WF	LF	WF	-	-	-	-
WF_CASN	WF	LF	LF	LF	LF	WF	WF	-	-	-
LF_LL	LF	LF	LF	LF	LF	LF	-	LF	LF	LF

3.2 Examples

3.2.1 C

```
NBLWordRoot *word;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  NBLWord *handle;

  /* Create a wait-free word object */
  word = NBLWordCreateWF_B();

  /* Initialize the memory words */
  handle=NBLWordGetHandle(word);
  NBLWordInit(handle,&(items[0]),1);
  NBLWordInit(handle,&(items[1]),2);
  NBLWordInit(handle,&(items[2]),3);
  NBLWordInit(handle,&(items[3]),4);

  /* Creating and running the threads ... */
  ...

  /* De-initialize the memory words */
```

```
  NBLWordDeinit(handle,&(items[0]));
  NBLWordDeinit(handle,&(items[1]));
  NBLWordDeinit(handle,&(items[2]));
  NBLWordDeinit(handle,&(items[3]));

  /* Freeing the memory that the word object used */
  NBLWordFree(word);
  word=NULL;
}

void Thread1()
{
  NBLWord *handle;
  /* Get the local handle to the word object */
  handle=NBLWordGetHandle(word);

  /* Atomically add the value 3 to items[3] */
  NBLWordAdd(handle,&(items[3]),3);

  /* Free the local handle to the word object */
  NBLWordFreeHandle(handle);

}

void Thread2()
{
  int value;
  NBLWord *handle;
  /* Get the local handle to the word object */
  handle=NBLWordGetHandle(word);

  /* Read the value of items[3], i.e. 4 or 7 */
  value=NBLWordRead(handle,&(items[3]));

  /* Free the local handle to the word object */
  NBLWordFreeHandle(handle);

}

void Thread3()
...
```

3.2.2 C++

```
NBL::Word<int> *word;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a wait-free word object */
```

```
  word = NBL::Word<int>::CreateWF_B();

  /* Initialize the memory words */
  word->Init(&(items[0]),1);
  word->Init(&(items[1]),2);
  word->Init(&(items[2]),3);
  word->Init(&(items[3]),4);

  /* Creating and running the threads ... */
  ...

  /* De-initialize the memory words */
  word->Deinit(&(items[0]));
  word->Deinit(&(items[1]));
  word->Deinit(&(items[2]));
  word->Deinit(&(items[3]));

  /* Freeing the memory that the word object used */
  delete word;
  word=NULL;
}

void Thread1()
{
  /* Atomically add the value 3 to items[3] */
  word->Add(&(items[3]),3);

}

void Thread2()
{
  int value;
  /* Read the value of items[3], i.e. 4 or 7 */
  value=word->Read(handle,&(items[3]));

}

void Thread3()
...
```

3.3 Creation

The following functions are used to create a new word object.

3.3.1 NBLWordCreateWF_B

Creates a new instance of a word object using a wait-free implementation. The implementation supports basic operations, i.e. Read, Write, Add, Swap, Op and CAS.

Syntax C

```
NBLWordRoot* NBLWordCreateWF_B(
);
```

Syntax C++

```
template class <>
static NBL::Word<int>* NBL::Word<int>::CreateWF_B(
);
```

Parameters

Return Values

If successful, the function returns a pointer to a new instance of a word object. Otherwise, it returns NULL.

Remarks

The operations INIT and DEINIT are not necessary to use for affected memory words in this implementation.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.3.2 NBLWordCreateWF_CASN

Creates a new instance of a word object using a wait-free implementation. The implementation supports basic operations as well as conditional updates of a set of memory words, i.e. transactions. The implementation requires a reduced accuracy of values in each memory word used, with 31 bits for actual use, i.e. it supports integers with values ranging from -1073741824 to 1073741823. The supported operations are Read, Write, Add, Swap, Op, CAS and CASN.

Syntax C

```
NBLWordRoot* NBLWordCreateWF_CASN(
  int nrOfThreads,
  int nrOfWords
);
```

Syntax C++

```
template class <>
static NBL::Word<int>* NBL::Word<int>::CreateWF_CASN(
  int nrOfThreads = 8,
  int nrOfWords = 8
);
```

Parameters

nrOfThreads
[in] The maximum number of concurrent threads that will access the structure.

nrOfWords
[in] The maximum number of memory words that are used in transaction instructions (i.e. CASN).

On-line Parameters

Return Values

If successful, the function returns a pointer to a new instance of a word object. Otherwise, it returns NULL.

Remarks

The functions Init and Deinit must be used for each affected memory word before vs. after any memory words are used in this implementation. Values outside of the allowed range inside of 31 bits of accuracy will be truncated.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.3.3 NBLWordCreateLF_LL

Creates a new instance of a word object using a lock-free implementation. The implementation supports basic operations as well as surveillance of possible updates of memory words. The supported operations are Read, Write, Add, Swap, Op, CAS, LL, VL and SC.

Syntax C

```
NBLWordRoot* NBLWordCreateLF_LL(
  int nrOfNodes,
  int nrOfVariables
);
```

Syntax C++

```
template class <>
static NBL::Word<int>* NBL::Word<int>::CreateLF_LL(
  int nrOfVariables = 4,
  int nrOfNodes = 1000
);
```

Parameters

nrOfNodes
[in] The maximum number of memory blocks that can be used for storing contents in the memory words. This number roughly matches the number of memory words for use: $nrOfNodes = n + N^2 * (1 + v) * 2$, *where* n=nrMemoryWords, N=nrOfThreads, v=nrOfVariables

nrOfVariables
[in] The maximum number of surveillance variables that can be used by each thread (i.e. handle).

On-line Parameters

Return Values

If successful, the function returns a pointer to a new instance of a word object. Otherwise, it returns NULL.

Remarks

The functions Init and Deinit must be used for each affected memory word before vs. after any memory words are used in this implementation.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4 Operations

3.4.1 NBLWordInit

Initializes a memory word for use with the word object.

Syntax C

```
int NBLWordInit(
  NBLWord* handle,
  void* address,
  int value
);
```

Syntax C++

```
template class <>
bool NBL::Word<int>::Init(
  void* address,
  int value
);
```

Parameters

handle
[in] A pointer to a local handle of a word object instance.

address
[in] The address of the memory word to access.

value
[in] The initial content of the memory word.

Return Values

If successful, the memory word is initialized to the specified value, and the function returns true. Otherwise, there is lack of remaining available resources for the purpose, and the function returns false.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.2 NBLWordDeinit

De-initializes a memory word after use with the word object.

Syntax C

```
void NBLWordDeinit(
  NBLWord* handle,
  void* address
);
```

Syntax C++

```
template class <>
void NBL::Word<int>::Deinit(
  void* address
);
```

Parameters

handle
[in] A pointer to a local handle of a word object instance.

address
[in] The address of the memory word to access.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.3 NBLWordRead

Reads the content of a memory word.

Syntax C

```
int NBLWordRead(
  NBLWord* handle,
  void* address
);
```

Syntax C++

```
template class <>
int NBL::Word<int>::Read(
  void* address
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

address
: [in] The address of the memory word to access.

Return Values

The function returns the content of the memory word at the specified address.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.4 NBLWordWrite

Writes a value to the content of a memory word.

Syntax C

```
void NBLWordWrite(
  NBLWord* handle,
  void* address,
  int value
);
```

Syntax C++

```
template class <>
void NBL::Word<int>::Write(
  void* address,
  int value
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

address
: [in] The address of the memory word to access.

value
: [in] The value to be written to the memory word.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.5 NBLWordAdd

Adds a value to the content of a memory word.

Syntax C

```
int NBLWordAdd(
  NBLWord* handle,
  void* address,
  int value
);
```

Syntax C++

```
template class <>
int NBL::Word<int>::Add(
  void* address,
  int value
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

address
: [in] The address of the memory word to access.

value
: [in] The value to be added to the content of the memory word.

Return Values

The function adds the value to the content of the memory word, and returns the new written content.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.6 NBLWordSwap

Exchanges the content of a memory word.

Syntax C

```
int NBLWordSwap(
  NBLWord* handle,
  void* address,
  int next
);
```

Syntax C++

```
template class <>
int NBL::Word<int>::Swap(
  void* address,
  int next
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

address
: [in] The address of the memory word to access.

next
: [in] The new value of the memory word.

Return Values

The functions read the content of the memory word, writes the new value to the memory word, and returns the old content.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.7 NBLWordOp

Updates the content of a memory word using a custom operation.

Syntax C

```
int NBLWordOp(
  NBLWord* handle,
  void* address,
  int (*fn)(int old, int arg),
  int value
);
```

Syntax C++

```
template class <>
int NBL::Word<int>::Op(
  void* address,
  int (*fn)(int old, int arg),
  int value
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

address
: [in] The address of the memory word to access.

fn
: [in] The custom function to apply for the content of the memory word.

value
: [in] The value to be passed as second argument to the custom function.

Return Values

The content of the memory word will be passed as first argument to the custom function. The result of the custom function will be written to the memory word, which written content will also be returned by this function.

Remarks

As the specified arithmetic function may be called several times due to retried attempts of updating, it must not cause any side effects.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.8 NBLWordCAS

Conditionally updates the content of a memory word

Syntax C

```
int NBLWordCAS(
  NBLWord* handle,
  void* address,
  int old
  int next
);
```

Syntax C++

```
template class <>
bool NBL::Word<int>::CAS(
  void* address,
  int old
  int next
);
```

Parameters

handle
[in] A pointer to a local handle of a word object instance.

address
[in] The address of the memory word to access.

old
[in] The awaited value of the memory word.

next
[in] The new value of the memory word.

Return Values

If the content of the specified memory word exactly match the specified awaited value, the contents of the memory word is updated with the specified new value, and the function returns true. Otherwise, the function returns false.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.9 NBLWordCASN

Conditionally updates the contents of a collection of memory words.

Syntax C

```
int NBLWordCASN(
  NBLWord* handle,
  int n,
  void* addresses[],
  int olds[],
  int nexts[]
);
```

Syntax C++

```
template class <>
bool NBL::Word<int>::CASN(
  int n,
  void* addresses[],
  int olds[],
  int nexts[]
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

n
: [in] The number of memory words to conditionally update.

addresses
: [in] An array containing the address for each specific memory word to access.

olds
: [in] An array containing the awaited values of each specific memory word.

nexts
: [in] An array containing the new values of each specific memory word.

Return Values

If the contents of all of the specified memory words exactly match the specified awaited values, the contents of the memory words are updated with the specified new values, and the function returns true.

Otherwise if any memory word does not match the specified awaited value, the function returns false.

Remarks

Although not required, for performance reasons it is desirable that the specified memory addresses are sorted in ascending order, i.e. lowest addresses first.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.10 NBLWordLL

Reads the content of a memory word, and starts surveillance for further updates.

Syntax C

```
int NBLWordLL(
  NBLWord* handle,
  int nr,
  void* address
);
```

Syntax C++

```
template class <>
int NBL::Word<int>::LL(
  int nr,
  void* address
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

nr
: [in] Index of the surveillance variable of interest.

address
: [in] The address of the memory word to access.

Return Values

The functions return the content of the memory word at the specified address and starts surveillance through the specified variable.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.11 NBLWordVL

Checks if the content of a memory word has been updated since the start of surveillance.

Syntax C

```
int NBLWordVL(
  NBLWord* handle,
  int nr,
  void* address
);
```

Syntax C++

```
template class <>
bool NBL::Word<int>::VL(
  int nr,
  void* address
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

nr
: [in] Index of the surveillance variable of interest.

address
: [in] The address of the memory word to access.

Return Values

If the memory word has not been updated since the start of the surveillance through the specified variable, the function returns true. Otherwise, the function returns false.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

3.4.12 NBLWordSC

Updates the content of a memory word only if the content has not been updated since the start of the surveillance.

Syntax C

```
int NBLWordSC(
  NBLWord* handle,
  int nr,
  void* address,
  int value
);
```

Syntax C++

```
template class <>
bool NBL::Word<int>::SC(
  int nr,
  void* address,
  int value
);
```

Parameters

handle
: [in] A pointer to a local handle of a word object instance.

nr
: [in] Index of the surveillance variable of interest.

address
: [in] The address of the memory word to access.

value
: [in] The value to conditionally write to the memory word.

Return Values

If the memory word has not been updated since the start of the surveillance through the specified variable, the value is written to the memory word and the function returns true. Otherwise, the function returns false.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 4

Memory Management

This is a collection of functions accessing several implementations of memory management schemes, i.e. handle memory allocation and garbage collection facilities.

- Three Lock-Free implementations offering strong garbage collection, limited number of local references, and bounded memory usage.
- Three Lock-Free implementations offering strong garbage collection, unlimited number of local references, and unbounded memory usage.
- Three Lock-Free implementations offering medium garbage collection, limited number of local references, and bounded memory usage.
- Three Lock-Free implementations offering weak garbage collection, limited number of local references, and bounded memory usage.
- A Wait-Free implementation offering strong garbage collection, unlimited number of local references, and unbounded memory usage.

The terms bounded versus unbounded memory usage just refers to the amount of memory needed for a certain number of elements in the targeted data structure, i.e. with bounded memory usage it can be determined in advance exactly how much memory is needed for a certain number of elements. The data structures are still fully dynamic and can contain as many elements as the size of the systems memory permits.

4.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

4.1.1 C

NBLMemoryCreateLF_SLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.1.

NBLMemoryCreateLF_CSLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.2.

NBLMemoryCreateLF_HSLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.3.

NBLMemoryCreateLF_SUU Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.4.

NBLMemoryCreateLF_CSUU Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.5.

NBLMemoryCreateLF_HSUU Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.6.

NBLMemoryCreateLF_MLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.7.

NBLMemoryCreateLF_CMLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.8.

NBLMemoryCreateLF_HMLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.9.

NBLMemoryCreateLF_WLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.10.

NBLMemoryCreateLF_CWLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.11.

NBLMemoryCreateLF_HWLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.12.

NBLMemoryCreateWF_SUU Creates a new instance of a memory manager object using a wait-free implementation supporting fixed-size blocks. See Section 4.2.13.

NBLMemoryAllocBlock Allocates a new memory block of fixed size. See Section 4.3.1.

NBLMemoryAllocClass Allocates a new memory block of specified size-class. See Section 4.3.2.

NBLMemoryAllocSize Allocates a new memory block of arbitrary size. See Section 4.3.3.

NBLMemoryDeleteBlock Frees a memory block when possible. See Section 4.3.4.

NBLMemoryDeRefLink Dereferences a shared memory pointer. See Section 4.3.5.

NBLMemoryCopyRef Copies a safe reference. See Section 4.3.6.

NBLMemoryReleaseRef Releases a safe reference. See Section 4.3.7.

NBLMemoryStoreRef Stores a reference in a shared memory pointer. See Section 4.3.8.

NBLMemoryCASRef Atomically updates a reference in a shared memory pointer. See Section 4.3.9.

4.1.2 C++

```
namespace NBL {
  template <typename T> class Memory {
    // Constructors
    static Memory<T>* CreateLF_SLB(int nrOfBlocks, int
        nrLocalRefs);
    static Memory<T>* CreateLF_SUU(int nrOfBlocks);
    static Memory<T>* CreateLF_MLB(int nrOfBlocks, int
        nrLocalRefs);
    static Memory<T>* CreateLF_WLB(int nrOfBlocks, int
        nrLocalRefs);
    static Memory<T>* CreateWF_SUU(int nrOfThreads, int
        nrOfBlocks);
    static Memory<T>* CreateLF_CSLB(NBLMemorySizeClass *
        sizeClasses, int nrSizeClasses, int nrLocalRefs);
```

```
    static Memory<T>* CreateLF_CSUU(NBLMemorySizeClass *
        sizeClasses, int nrSizeClasses);
    static Memory<T>* CreateLF_CMLB(NBLMemorySizeClass *
        sizeClasses, int nrSizeClasses, int nrLocalRefs);
    static Memory<T>* CreateLF_CWLB(NBLMemorySizeClass *
        sizeClasses, int nrSizeClasses, int nrLocalRefs);
    static Memory<T>* CreateLF_HSLB(int heapSize, int
        nrLocalRefs);
    static Memory<T>* CreateLF_HSUU(int heapSize);
    static Memory<T>* CreateLF_HMLB(int heapSize, int
        nrLocalRefs);
    static Memory<T>* CreateLF_HWLB(int heapSize, int
        nrLocalRefs);
    // Operations
    T* AllocBlock();
    void DelBlock(T* block);
    T* DeRefLink(T** link);
    void CopyRef(T* block);
    void ReleaseRef(T* \&block);
    void StoreRef(T** link, T* block);
    bool CASRef(T** link, T* old, T* block);
  };
};
```

NBL::Memory::CreateLF_SLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.1.

NBL::Memory::CreateLF_CSLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.2.

NBL::Memory::CreateLF_HSLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.3.

NBL::Memory::CreateLF_SUU Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.4.

NBL::Memory::CreateLF_CSUU Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.5.

NBL::Memory::CreateLF_HSUU Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.6.

NBL::Memory::CreateLF_MLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.7.

NBL::Memory::CreateLF_CMLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.8.

NBL::Memory::CreateLF_HMLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.9.

NBL::Memory::CreateLF_WLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.10.

NBL::Memory::CreateLF_CWLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.11.

NBL::Memory::CreateLF_HWLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.12.

NBL::Memory::CreateWF_SUU Creates a new instance of a memory manager object using a wait-free implementation supporting fixed-size blocks. See Section 4.2.13.

NBL::Memory::AllocBlock Allocates a new memory block of fixed size. See Section 4.3.1.

NBL::Memory::AllocClass Allocates a new memory block of specified size-class. See Section 4.3.2.

NBL::Memory::AllocSize Allocates a new memory block of arbitrary size. See Section 4.3.3.

NBL::Memory::DeleteBlock Frees a memory block when possible. See Section 4.3.4.

NBL::Memory::DeRefLink Dereferences a shared memory pointer. See Section 4.3.5.

NBL::Memory::CopyRef Copies a safe reference. See Section 4.3.6.

NBL::Memory::ReleaseRef Releases a safe reference. See Section 4.3.7.

NBL::Memory::StoreRef Stores a reference in a shared memory pointer. See Section 4.3.8.

NBL::Memory::CASRef Atomically updates a reference in a shared memory pointer. See Section 4.3.9.

4.2 Creation

The following functions are used to create a new memory manager.

NBLMemoryCreateLF_SLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.1.

NBLMemoryCreateLF_CSLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.2.

NBLMemoryCreateLF_HSLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.3.

NBLMemoryCreateLF_SUU Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.4.

NBLMemoryCreateLF_CSUU Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.5.

NBLMemoryCreateLF_HSUU Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.6.

NBLMemoryCreateLF_MLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.7.

NBLMemoryCreateLF_CMLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.8.

NBLMemoryCreateLF_HMLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.9.

NBLMemoryCreateLF_WLB Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks. See Section 4.2.10.

NBLMemoryCreateLF_CWLB Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks. See Section 4.2.11.

NBLMemoryCreateLF_HWLB Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks. See Section 4.2.12.

NBLMemoryCreateWF_SUU Creates a new instance of a memory manager object using a wait-free implementation supporting fixed-size blocks. See Section 4.2.13.

4.2.1 NBLMemoryCreateLF_SLB

Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_SLB(
  int nrOfBlocks,
  int sizeOfBlocks,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_SLB(
  int nrOfBlocks = 1000,
  int nrLocalRefs = 8
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used.

sizeOfBlocks
[in] The size in bytes of each memory block.

nrLocalRefs
[in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_REDUCE_CHAIN
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

The implementation also supports a user-defined callback function to be called whenever the memory manager is about to free a memory block which may hinder other memory blocks from being freed. The purpose of this function is to make sure that all safe references that is stored in links within this memory block only references other memory blocks that are not about to being freed. The syntax of this callback function is as follows:

```
void NBLMemoryReduceChainCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.2 NBLMemoryCreateLF_CSLB

Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_CSLB(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_CSLB(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses,
  int nrLocalRefs = 8
);
```

Parameters

sizeClasses
: [in] Definitions of the size classes to use.

nrSizeClasses
: [in] The number of size classes to use.

nrLocalRefs
: [in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC
PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_REDUCE_CHAIN
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

The implementation also supports a user-defined callback function to be called whenever the memory manager is about to free a memory block which may hinder other memory blocks from being freed. The purpose of this function is to make sure that all safe references that is stored in links within this memory block only references other memory blocks that are not about to being freed. The syntax of this callback function is as follows:

```
void NBLMemoryReduceChainCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.3 NBLMemoryCreateLF_HSLB

Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_HSLB(
  int heapSize,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_HSLB(
  int heapSize,
  int nrLocalRefs = 8
);
```

Parameters

heapSize
[in] The number of bytes to use for the heap.

nrLocalRefs
[in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC
PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_REDUCE_CHAIN
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

The implementation also supports a user-defined callback function to be called whenever the memory manager is about to free a memory block which may hinder other memory blocks from being freed. The purpose of this function is to make sure that all safe references that is stored in links within this memory block only references other memory blocks that are not about to being freed. The syntax of this callback function is as follows:

```
void NBLMemoryReduceChainCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.4 NBLMemoryCreateLF_SUU

Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is unlimited. The number of logically deleted but still not reclaimed memory blocks is unbounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_SUU(
  int nrOfBlocks,
  int sizeOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_SUU(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used.

sizeOfBlocks
[in] The size in bytes of each memory block.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC
PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.5 NBLMemoryCreateLF_CSUU

Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is unlimited. The number of logically deleted but still not reclaimed memory blocks is unbounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_CSUU(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_CSUU(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses
);
```

Parameters

sizeClasses
[in] Definitions of the size classes to use.

nrSizeClasses
[in] The number of size classes to use.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC
PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.6 NBLMemoryCreateLF_HSUU

Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is unlimited. The number of logically deleted but still not reclaimed memory blocks is unbounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_HSUU(
  int heapSize
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_HSUU(
  int heapSize
);
```

Parameters

heapSize
[in] The number of bytes to use for the heap.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC
PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.7 NBLMemoryCreateLF_MLB

Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks.

The implementation offers medium strong garbage collection facilities and can thus only under certain circumstances allow safe dereferencing of links that are within memory blocks which has been logically deleted. The number of local safe references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_MLB(
  int nrOfBlocks,
  int sizeOfBlocks,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_MLB(
  int nrOfBlocks = 1000,
  int nrLocalRefs = 8
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used.

sizeOfBlocks
[in] The size in bytes of each memory block.

nrLocalRefs
[in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.8 NBLMemoryCreateLF_CMLB

Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks.

The implementation offers medium strong garbage collection facilities and can thus only under certain circumstances allow safe dereferencing of links that are within memory blocks which has been logically deleted. The number of local safe references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_CMLB(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_CMLB(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses,
  int nrLocalRefs = 8
);
```

Parameters

sizeClasses
: [in] Definitions of the size classes to use.

nrSizeClasses
: [in] The number of size classes to use.

nrLocalRefs
: [in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.9 NBLMemoryCreateLF_HMLB

Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks.

The implementation offers medium strong garbage collection facilities and can thus only under certain circumstances allow safe dereferencing of links that are within memory blocks which has been logically deleted. The number of local safe references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_HMLB(
  int heapSize,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_HMLB(
  int heapSize,
  int nrLocalRefs = 8
);
```

Parameters

heapSize
: [in] The number of bytes to use for the heap.

nrLocalRefs
: [in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.10 NBLMemoryCreateLF_WLB

Creates a new instance of a memory manager object using a lock-free implementation supporting fixed-size blocks.

The implementation offers weak garbage collection facilities and thus do not allow safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_WLB(
  int nrOfBlocks,
  int sizeOfBlocks,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_WLB(
  int nrOfBlocks = 1000,
  int nrLocalRefs = 8
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used.

sizeOfBlocks
[in] The size in bytes of each memory block.

nrLocalRefs
[in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.11 NBLMemoryCreateLF_CWLB

Creates a new instance of a memory manager object using a lock-free implementation supporting multi-sized blocks.

The implementation offers weak garbage collection facilities and thus do not allow safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_CWLB(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_CWLB(
  NBLMemorySizeClass *sizeClasses,
  int nrSizeClasses,
  int nrLocalRefs = 8
);
```

Parameters

sizeClasses
[in] Definitions of the size classes to use.

nrSizeClasses
[in] The number of size classes to use.

nrLocalRefs
[in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.12 NBLMemoryCreateLF_HWLB

Creates a new instance of a memory manager object using a lock-free implementation supporting any-sized blocks.

The implementation offers weak garbage collection facilities and thus do not allow safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is limited. The number of logically deleted but still not reclaimed memory blocks is bounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateLF_HWLB(
  int heapSize,
  int nrLocalRefs
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateLF_HWLB(
  int heapSize,
  int nrLocalRefs = 8
);
```

Parameters

heapSize
[in] The number of bytes to use for the heap.

nrLocalRefs
[in] The maximum number of safe local references that can be used per handle.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.2.13 NBLMemoryCreateWF_SUU

Creates a new instance of a memory manager object using a wait-free implementation supporting fixed-size blocks.

The implementation offers strong garbage collection facilities and thus allows safe dereferencing of links that are within memory blocks which has been logically deleted. The number of safe local references that can be used per handle is unlimited. The number of logically deleted but still not reclaimed memory blocks is unbounded.

Syntax C

```
NBLMemoryRoot* NBLMemoryCreateWF_SUU(
  int nrOfThreads,
  int nrOfBlocks,
  int sizeOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Memory<T>* NBL::Memory<T>::CreateWF_SUU(
  int nrOfThreads = 16,
  int nrOfBlocks = 1000,
);
```

Parameters

nrOfThreads
[in] The maximum number of concurrent threads that will access the structure.

nrOfBlocks
[in] The maximum number of memory blocks that can be used.

sizeOfBlocks
[in] The size in bytes of each memory block.

On-line Parameters

PARAM_MEM_OVERHEAD_SIZE
PARAM_MEM_FIXED_ALLOC
PARAM_MEM_CB_RELEASE_REFS
PARAM_MEM_CB_ARG1

For detailed parameter descriptions see Section 2.4.1.

Return Values

If successful, the function returns a pointer to a new instance of a memory manager object. Otherwise, it returns NULL.

Remarks

The implementation supports a user-defined callback function to be called whenever the memory manager is about to free a memory block. The purpose of this function is to release all recursive safe references that are stored in links within this memory block. The syntax of this callback function is as follows:

```
void NBLMemoryReleaseRefsCB(void *arg1, void *block);
```

where arg1 is a fixed user-defined argument and block is the memory block of interest.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3 Operations

4.3.1 NBLMemoryAllocBlock

Allocates a new memory block.

Syntax C

```
void * NBLMemoryAllocBlock(
  NBLMemory* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::Memory<T>::AllocBlock(
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

Return Values

If successful, the function returns a safe pointer to a new allocated memory block. Otherwise the memory pool of the memory manager was empty, i.e. out of memory.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.2 NBLMemoryAllocClass

Allocates a new memory block of selected size-class.

Syntax C

```
void * NBLMemoryAllocClass(
  NBLMemory* handle,
  int sizeClass
);
```

Syntax C++

```
template <typename T> class
T* NBL::Memory<T>::AllocClass(
  int sizeClass
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

Return Values

If successful, the function returns a safe pointer to a new allocated memory block. Otherwise the memory pool of the memory manager was empty, i.e. out of memory.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.3 NBLMemoryAllocSize

Allocates a new memory block of arbitrary size.

Syntax C

```
void * NBLMemoryAllocSize(
  NBLMemory* handle,
  int size
);
```

Syntax C++

```
template <typename T> class
T* NBL::Memory<T>::AllocSize(
  int size
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

Return Values

If successful, the function returns a safe pointer to a new allocated memory block. Otherwise the memory pool of the memory manager was empty, i.e. out of memory.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.4 NBLMemoryDeleteBlock

Frees a memory block when possible, i.e. when no threads or other blocks keep safe references to it.

Syntax C

```
void NBLMemoryDeleteBlock(
  NBLMemory* handle,
  void* block
);
```

Syntax C++

```
template <typename T> class
void NBL::Memory<T>::DeleteBlock(
  T* block
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

block
: [in] Pointer to the block which memory should be freed.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.5 NBLMemoryDeRefLink

Dereferences a shared memory pointer.

Syntax C

```
void* NBLMemoryDeRefLink(
  NBLMemory* handle,
  void** link
);
```

Syntax C++

```
template <typename T> class
T* NBL::Memory<T>::DeRefLink(
  T** link
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

link
: [in] Pointer to a shared link which pointer to a memory block should be de-referenced.

Return Values

If successful, the function returns a safe pointer to the memory block pointed to by the shared link. Otherwise the link was invalid, i.e. NULL.

Remarks

Depending on the implementation chosen, this function does or does not allow safe dereferencing of links that are within memory blocks which has been logically deleted.

The safe references obtained are local and can thus not be directly shared among threads. To share a safe reference this has to be written to a link and be correspondingly de-referenced.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

NBLMemoryCopyRef *Creates another safe reference to the same memory block. See Section 4.3.6.*

NBLMemoryReleaseRef *Releases a safe reference when it is no longer needed. See Section 4.3.7.*

4.3.6 NBLMemoryCopyRef

Copies a safe reference.

Syntax C

```
void* NBLMemoryCopyRef(
  NBLMemory* handle,
  void* block
);
```

Syntax C++

```
template <typename T> class
T* NBL::Memory<T>::CopyRef(
  T* block
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

block
: [in] The safe reference which should be copied.

Return Values

If successful, the function returns a new safe pointer to a memory block. Otherwise it was not possible to create a new reference to the memory block, or the pointer was invalid, i.e. NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.7 NBLMemoryReleaseRef

Releases a safe reference to a memory block.

Syntax C

```
void NBLMemoryReleaseRef(
  NBLMemory* handle,
  void* block
);
```

Syntax C++

```
template <typename T> class
void NBL::Memory<T>::ReleaseRef(
  T* \&block
);
```

Parameters

handle
[in] A pointer to a local handle of a memory manager object instance.

block
[in] The safe reference which should be released.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.8 NBLMemoryStoreRef

Stores a reference to a memory block in a shared memory pointer.

Syntax C

```
void NBLMemoryStoreRef(
  NBLMemory* handle,
  void** link,
  void* block
);
```

Syntax C++

```
template <typename T> class
void NBL::Memory<T>::StoreRef(
  T** link,
  T* block
);
```

Parameters

handle
[in] A pointer to a local handle of a memory manager object instance.

link
[in] Pointer to a shared link which pointer to a memory block should be updated.

block
[in] A safe reference to a memory block.

Return Values

Remarks

Before first usage the corresponding link should have been initiated with NULL.

There must not be any possibility of concurrent updates to the corresponding link at the time this function is called.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

4.3.9 NBLMemoryCASRef

Atomically updates a reference in a shared memory pointer.

Syntax C

```
int NBLMemoryCASRef(
  NBLMemory* handle,
  void** link,
  void* old,
  void* new
);
```

Syntax C++

```
template <typename T> class
bool NBL::Memory<T>::CASRef(
  T** link,
  T* old,
  T* new
);
```

Parameters

handle
: [in] A pointer to a local handle of a memory manager object instance.

link
: [in] Pointer to a shared link which pointer to a memory block should be updated.

old
: [in] A safe reference to a memory block.

new
: [in] A safe reference to a memory block.

Return Values

If the link was pointing to the old reference at the time of call, the link is atomically updated to point to the new reference and the functions returns true. Otherwise, the function returns false.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 5

Shared Stack

The Stack abstract data type is a collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are *Push* (add to the top) and *Pop* (remove from the top). Pop returns the item removed. The data structure is also known as a ”last-in, first-out” or LIFO buffer.

The implementations available of a shared stack are:

- A Lock-Free implementation offering bounded memory usage.
- A Lock-Free implementation offering unbounded memory usage.
- A Lock-Based implementation.

All of the available implementations of a shared stack offers the same high-level semantics.

Definition 1 *We denote with Q_t the abstract state of a stack at the time t. $Q_t = [v_1, \ldots, v_n]$ is viewed as an list of values v, where $|Q_t| \geq 0$. The operations that can be performed on the stack are* Push*, and* Pop*. The time t_1 is defined as the time just before the operation of interest takes place, and the time t_2 is defined as the time just after the same operation takes place. In the following expressions that define the high-level semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$Q_{t_1} : \mathbf{Push(v_1)}, Q_{t_2} = [v_1] + Q_{t_1} \tag{5.1}$$

$$Q_{t_1} = \emptyset : \mathbf{Pop}() = \bot, Q_{t_2} = \emptyset \tag{5.2}$$

$$Q_{t_1} = [v_1] + Q_1 : \mathbf{Pop}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{5.3}$$

The detailed semantics of these operations are described in the respective detailed description of each individual operation.

5.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

5.1.1 C

NBLStackCreateLF_B Creates a new instance of a stack object using a lock-free implementation. See Section 5.3.1.

NBLStackCreateLF_U Creates a new instance of a stack object using a lock-free implementation. See Section 5.3.2.

NBLStackCreateLB Creates a new instance of a stack object using a lock-based implementation. See Section 5.3.3.

NBLStackPush Pushes a new item on the stack. See Section 5.4.1.

NBLStackPop Pops an item from the stack. See Section 5.4.2.

5.1.2 C++

```
namespace NBL {
  template <typename T> class Stack {
    // Constructors
    static Stack<T>* CreateLF_B(int nrOfBlocks);
    static Stack<T>* CreateLF_U(int nrOfBlocks);
    static Stack<T>* CreateLB();
    // Operations
    bool Push(T *item);
    T *Pop();
  };
};
```

NBL::Stack::CreateLF_B Creates a new instance of a stack object using a lock-free implementation. See Section 5.3.1.

NBL::Stack::CreateLF_U Creates a new instance of a stack object using a lock-free implementation. See Section 5.3.2.

NBL::Stack::CreateLB Creates a new instance of a stack object using a lock-based implementation. See Section 5.3.3.

NBL::Stack::Push Pushes a new item on the stack. See Section 5.4.1.

NBL::Stack::Pop Pops an item from the stack. See Section 5.4.2.

5.2 Examples

5.2.1 C

```
NBLStackRoot *stack;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free stack data object, with maximum space
      for appr. 100 items */
  stack = NBLStackCreateLF_B(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the stack used */
  NBLStackFree(stack);
  stack=NULL;
}

void Thread1()
{
  NBLStack *handle;
  /* Get the local handle to the shared stack */
  handle=NBLStackGetHandle(stack);

  /* Push an item on the stack, i.e. pushing items[3] */
  NBLStackPush(handle,&(items[3]));

  /* Free the local handle to the shared stack */
  NBLStackFreeHandle(handle);

}

void Thread2()
{
  int *item;
  NBLStack *handle;
  /* Get the local handle to the shared stack */
  handle=NBLStackGetHandle(stack);

  /* Push another item on the stack, i.e. pushing items[2] */
  NBLStackPush(handle,&(items[2]));

  /* Poping an item back from the stack i.e. we should get a
      pointer to items[2] or items[3] depending on which thread
      was the fastest */
  item = NBLStackPop(handle);
```

```
  /* Free the local handle to the shared stack */
  NBLStackFreeHandle(handle);
}

void Thread3()
...
```

5.2.2 C++

```
NBLStack<int> *stack;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free stack data object, with maximum space
     for appr. 100 items */
  stack = NBL::Stack<int>::CreateLF_B(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the stack used */
  delete stack;
  stack=NULL;
}

void Thread1()
{
  /* Push an item on the stack, i.e. pushing items[3] */
  stack->Push(&(items[3]));

}

void Thread2()
{
  int *item;

  /* Push another item on the stack, i.e. pushing items[2] */
  stack->Push(&(items[2]));

  /* Poping an item back from the stack i.e. we should get a
     pointer to items[2] or items[3] depending on which thread
     was the fastest */
  item = stack->Pop();


}

void Thread3()
```

```
...
```

5.3 Creation

The following functions are used to create a new stack shared data structure.

NBLStackCreateLF_B Creates a new instance of a stack object using a lock-free implementation. See Section 5.3.1.

NBLStackCreateLF_U Creates a new instance of a stack object using a lock-free implementation. See Section 5.3.2.

NBLStackCreateLB Creates a new instance of a stack object using a lock-based implementation. See Section 5.3.3.

5.3.1 NBLStackCreateLF_B

Creates a new instance of a stack object using a lock-free implementation.

The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLStackRoot* NBLStackCreateLF_B(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Stack<T>* NBL::Stack<T>::CreateLF_B(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the stack. This measures approximately the number of items that can be stored into the stack: $nrOfBlocks = n + N^2 * 2 + N$, *where* n=nrOfItems, N=nrOfThreads

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a stack object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

5.3.2 NBLStackCreateLF_U

Creates a new instance of a stack object using a lock-free implementation.

The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLStackRoot* NBLStackCreateLF_U(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Stack<T>* NBL::Stack<T>::CreateLF_U(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the stack. This measures approximately the number of items that can be stored into the stack: $nrOfBlocks = n + N * 2$, *where* n=nrOfItems, N=nrOfThreads

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a stack object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

5.3.3 NBLStackCreateLB

Creates a new instance of a stack object using a lock-based implementation.

Syntax C

```
NBLStackRoot* NBLStackCreateLB(
);
```

Syntax C++

```
template <typename T> class
static NBL::Stack<T>* NBL::Stack<T>::CreateLB(
);
```

Parameters

Return Values

If successful, the function returns a pointer to a new instance of a stack object. Otherwise, it returns NULL.

Remarks

The locks are based on spin-locks.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

5.4 Operations

5.4.1 NBLStackPush

Pushes a new item on the stack.

Syntax C

```
int NBLStackPush(
  NBLStack* handle,
  void* item
);
```

Syntax C++

```
template <typename T> class
bool NBL::Stack<T>::Push(
  T* item
);
```

Parameters

handle
: [in] A pointer to a local handle of a stack object instance.

item
: [in] Pointer to the item to push on the stack. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the stack was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all pushed items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

5.4.2 NBLStackPop

Pops an item from the stack.

Syntax C

```
void* NBLStackPop(
   NBLStack* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::Stack<T>::Pop(
);
```

Parameters

handle
[in] A pointer to a local handle of a stack object instance.

Return Values

If successful, the function returns a pointer to the item that was on the top of the stack at some point in time and directly afterwards removed from the stack. Otherwise the stack was empty at some point in time, and the function returns NULL.

Remarks

The actual item is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 6

Shared Queue

The Queue abstract data type is a collection of items in which only the earliest added item may be accessed. Basic operations are *Enqueue* (add to the tail) and *Dequeue* (remove from the head). Dequeue returns the item removed. The data structure is also known as a "first-in, first-out" or FIFO buffer.

The implementations available of a shared queue are:

- A Wait-Free implementation offering single enqueuer, and single dequeuer support.
- A Lock-Free implementation offering dynamic structure, and bounded memory usage.
- A Lock-Free implementation offering static structure, and bounded memory usage.
- A Lock-Free implementation offering dynamic structure, and unbounded memory usage.
- A Lock-Based implementation.

All of the available implementations of a shared queue offers the same high-level semantics.

Definition 2 *We denote with Q_t the abstract state of a queue at the time t. $Q_t = [v_1, \ldots, v_n]$ is viewed as an list of values v, where $|Q_t| \geq 0$. The operations that can be performed on the queue are* Enqueue, *and* Dequeue. *The time t_1 is defined as the time just before the operation of interest takes place, and the time t_2 is defined as the time just after the same operation takes place. In the following expressions that define the high-level semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$Q_{t_1} : \mathbf{Enqueue}(\mathbf{v_1}), Q_{t_2} = [v_1] + Q_{t_1} \quad (6.1)$$

$$Q_{t_1} = \emptyset : \mathbf{Dequeue}() = \bot, Q_{t_2} = \emptyset \quad (6.2)$$

$$Q_{t_1} = Q_1 + [v_1] : \mathbf{Dequeue}() = \mathbf{v_1}, Q_{t_2} = Q_1 \quad (6.3)$$

The detailed semantics of these operations are described in the respective detailed description of each individual operation.

6.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

6.1.1 C

NBLQueueCreateWF_SS Creates a new instance of a queue object using a wait-free implementation. See Section 6.3.1.

NBLQueueCreateLF_DB Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.2.

NBLQueueCreateLF_SB Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.3.

NBLQueueCreateLF_DU Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.4.

NBLQueueCreateLB Creates a new instance of a queue object using a lock-based implementation. See Section 6.3.5.

NBLQueueEnqueue Puts a new item on top of the queue. See Section 6.4.1.

NBLQueueDequeue Removes an item from bottom of the queue. See Section 6.4.2.

NBLQueueIsEmpty Answers whether the queue is empty or not. See Section 6.4.3.

NBLQueueSize Estimates the current number of items in the queue. See Section 6.4.4.

6.1.2 C++

```
namespace NBL {
  template <typename T> class Queue {
    // Constructors
    static Queue <T>* CreateLF_DB(int nrOfBlocks);
    static Queue <T>* CreateLF_DU(int nrOfBlocks);
    static Queue<T>* CreateLF_SB(int nrNodes);
    static Queue<T>* CreateWF_SS(int nrNodes);
    static Queue <T>* CreateLB();
    // Operations
    bool Enqueue(T *item);
    T *Dequeue();
    int Size();
    bool IsEmpty();
  };
};
```

NBL::Queue::CreateWF_SS Creates a new instance of a queue object using a wait-free implementation. See Section 6.3.1.

NBL::Queue::CreateLF_DB Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.2.

NBL::Queue::CreateLF_SB Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.3.

NBL::Queue::CreateLF_DU Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.4.

NBL::Queue::CreateLB Creates a new instance of a queue object using a lock-based implementation. See Section 6.3.5.

NBL::Queue::Enqueue Puts a new item on top of the queue. See Section 6.4.1.

NBL::Queue::Dequeue Removes an item from bottom of the queue. See Section 6.4.2.

NBL::Queue::IsEmpty Answers whether the queue is empty or not. See Section 6.4.3.

NBL::Queue::Size Estimates the current number of items in the queue. See Section 6.4.4.

6.2 Examples

6.2.1 C

```
NBLQueueRoot *queue;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free queue data object, with maximum space
     for appr. 100 items */
  queue = NBLQueueCreateLF_DB(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the queue object used */
  NBLQueueFree(queue);
  queue=NULL;
}

void Thread1()
{
  NBLQueue *handle;
  /* Get the local handle to the shared queue */
  handle=NBLQueueGetHandle(queue);

  /* Enqueue an item on the queue, i.e. putting items[3] on the
      queue */
  NBLQueueEnqueue(handle,&(items[3]));

  /* Free the local handle to the shared queue */
  NBLQueueFreeHandle(handle);

}

void Thread2()
{
  int *item;
  NBLQueue *handle;
  /* Get the local handle to the shared queue */
  handle=NBLQueueGetHandle(queue);

  /* Enqueue another item on the queue, i.e. putting items[2]
     on the queue */
  NBLQueueEnqueue(handle,&(items[2]));

  /* Dequeueing an item from the queue i.e. we should get a
     pointer to items[3] or items[2] depending of which thread
     was the fastest */
  item = NBLQueueDequeue(handle);

  /* Free the local handle to the shared queue */
  NBLQueueFreeHandle(handle);
```

```
}

void Thread3()
...
```

6.2.2 C++

```
NBL::Queue<int> *queue;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free queue data object, with maximum space
     for appr. 100 items */
  queue = NBL::Queue<int>::CreateLF_DB(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the queue object used */
  delete queue;
  queue=NULL;
}

void Thread1()
{
  /* Enqueue an item on the queue, i.e. putting items[3] on the
      queue */
  queue->Enqueue(&(items[3]));

}

void Thread2()
{
  int *item;

  /* Enqueue another item on the queue, i.e. putting items[2]
     on the queue */
  queue->Enqueue(&(items[2]));

  /* Dequeueing an item from the queue i.e. we should get a
     pointer to items[3] or items[2] depending of which thread
     was the fastest */
  item = queue->Dequeue();

}

void Thread3()
...
```

6.3 Creation

The following functions are used to create a new queue shared data structure.

NBLQueueCreateWF_SS Creates a new instance of a queue object using a wait-free implementation. See Section 6.3.1.

NBLQueueCreateLF_DB Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.2.

NBLQueueCreateLF_SB Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.3.

NBLQueueCreateLF_DU Creates a new instance of a queue object using a lock-free implementation. See Section 6.3.4.

NBLQueueCreateLB Creates a new instance of a queue object using a lock-based implementation. See Section 6.3.5.

6.3.1 NBLQueueCreateWF_SS

Creates a new instance of a queue object using a wait-free implementation. The implementation supports limited concurrency, only allowing one enqueue operation concurrently with one dequeue operation.

The items are stored in a static structure that increases dynamically in size with increasing queue size. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLQueueRoot* NBLQueueCreateWF_SS(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Queue<T>* NBL::Queue<T>::CreateWF_SS(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
: [in] The initial number of memory blocks that are used for the implementation of the queue. This measures approximately the number of items that can be stored into the queue: $nrOfBlocks = n + 1$, *where* n=nrOfItems, N=nrOfThreads

Return Values

If successful, the function returns a pointer to a new instance of a queue object. Otherwise, it returns NULL.

Remarks

This implementation only supports two handles, one performing enqueue operations and the other performing dequeue operations, possibly concurrently.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.3.2 NBLQueueCreateLF_DB

Creates a new instance of a queue object using a lock-free implementation.

The items are stored in a dynamic structure. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLQueueRoot* NBLQueueCreateLF_DB(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Queue<T>* NBL::Queue<T>::CreateLF_DB(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory blocks that can be used for the implementation of the queue. This measures approximately the number of items that can be stored into the queue: $nrOfBlocks = n + N^2 * 4 + N$, *where* n=nrOfItems, N=nrOfThreads

Return Values

If successful, the function returns a pointer to a new instance of a queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.3.3 NBLQueueCreateLF_SB

Creates a new instance of a queue object using a lock-free implementation.

The items are stored in a static structure. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLQueueRoot* NBLQueueCreateLF_SB(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Queue<T>* NBL::Queue<T>::CreateLF_SB(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the queue. This measures approximately the number of items that can be stored into the queue: $nrOfBlocks = n$, *where* n=nrOfItems

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.3.4 NBLQueueCreateLF_DU

Creates a new instance of a queue object using a lock-free implementation. The items are stored in a dynamic structure. The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLQueueRoot* NBLQueueCreateLF_DU(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Queue<T>* NBL::Queue<T>::CreateLF_DU(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory blocks that can be used for the implementation of the queue. This measures approximately the number of items that can be stored into the queue: $nrOfBlocks \geq n + N * 5$, *where* n=nrOfItems, N=nrOfThreads

Return Values

If successful, the function returns a pointer to a new instance of a queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.3.5 NBLQueueCreateLB

Creates a new instance of a queue object using a lock-based implementation.

Syntax C

```
NBLQueueRoot* NBLQueueCreateLB(
);
```

Syntax C++

```
template <typename T> class
static NBL::Queue<T>* NBL::Queue<T>::CreateLB(
);
```

Parameters

Return Values

If successful, the function returns a pointer to a new instance of a queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.4 Operations

6.4.1 NBLQueueEnqueue

Puts a new item on top of the queue.

Syntax C

```
int NBLQueueEnqueue(
  NBLQueue* handle,
  void* item
);
```

Syntax C++

```
template <typename T> class
bool NBL::Queue<T>::Enqueue(
  void* item
);
```

Parameters

handle
: [in] A pointer to a local handle of a queue object instance.

item
: [in] Pointer to the item to push on the queue. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the queue was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all enqueued items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.4.2 NBLQueueDequeue

Removes an item from bottom of the queue.

Syntax C

```
void* NBLQueueDequeue(
  NBLQueue* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::Queue<T>::Dequeue(
);
```

Parameters

handle
: [in] A pointer to a local handle of a queue object instance.

Return Values

If successful, the function returns a pointer to the item that was on the head of the queue at some point in time and directly afterwards removed from the queue. Otherwise the queue was empty at some point in time, and the function returns NULL.

Remarks

The actual item is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.4.3 NBLQueueIsEmpty

Answers whether the queue is empty or not.

Syntax C

```
int NBLQueueIsEmpty(
  NBLQueue* handle
);
```

Syntax C++

```
template <typename T> class
bool NBL::Queue<T>::IsEmpty(
);
```

Parameters

handle
: [in] A pointer to a local handle of a queue object instance.

Return Values

If the queue does not contain any items, the function returns true. Otherwise the queue contains items, and the function returns false.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

6.4.4 NBLQueueSize

Estimates the current number of items in the queue.

Syntax C

```
int NBLQueueSize(
  NBLQueue* handle
);
```

Syntax C++

```
template <typename T> class
int NBL::Queue<T>::Size(
);
```

Parameters

handle
[in] A pointer to a local handle of a queue object instance.

Return Values

The function returns an estimation of the number of items currently stored in the queue.

Remarks

This operation might not be linearizable in all implementations, and may thus return an estimate which is not accurate according to time.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 7

Shared Deque

The Deque (or doubly-ended queue) abstract data type is a combination of the stack and the queue abstract data types. The data structure is a collection of items in which the earliest as well as the latest added item may be accessed. Basic operations are *PushLeft* (add to the head), *PopLeft* (remove from the head), *PushRight* (add to the tail), and *PopRight* (remove from the tail). PopLeft and PopRight returns the item removed.

The implementations available of a shared deque are:

- A Lock-Free implementation offering high parallelism, and bounded memory usage.
- A Lock-Free implementation offering high parallelism, and unbounded memory usage.
- A Lock-Free implementation offering low parallelism, and bounded memory usage.
- A Lock-Based implementation.

The terms bounded versus unbounded memory usage just refers to the amount of memory needed for a certain number of elements in the targeted data structure, i.e. with bounded memory usage it can be determined in advance exactly how much memory is needed for a certain number of elements. The data structures are still fully dynamic and can contain as many elements as the size of the systems memory permits.

All of the available implementations of a shared deque offers the same high-level semantics.

Definition 3 *We denote with Q_t the abstract state of a deque at the time t. $Q_t = [v_1, \ldots, v_n]$ is viewed as an list of values v, where $|Q_t| \geq 0$. The operations that can be performed on the deque are* PushLeft, PushRight, PopLeft *and* PopRight*. The time t_1 is defined as the time just before the operation of interest takes place, and the time t_2 is defined as the time just after the same operation takes place. In the following expressions that define the high-level semantics of our operations, the syntax is $S_1 : O_1, S_2$, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:*

$$Q_{t_1} : \mathbf{PushLeft}(\mathbf{v_1}), Q_{t_2} = [v_1] + Q_{t_1} \tag{7.1}$$

$$Q_{t_1} : \mathbf{PushRight}(\mathbf{v_1}), Q_{t_2} = Q_{t_1} + [v_1] \tag{7.2}$$

$$Q_{t_1} = \emptyset : \mathbf{PopLeft}() = \bot, Q_{t_2} = \emptyset \tag{7.3}$$

$$Q_{t_1} = [v_1] + Q_1 : \mathbf{PopLeft}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{7.4}$$

$$Q_{t_1} = \emptyset : \mathbf{PopRight}() = \bot, Q_{t_2} = \emptyset \tag{7.5}$$

$$Q_{t_1} = Q_1 + [v_1] : \mathbf{PopRight}() = \mathbf{v_1}, Q_{t_2} = Q_1 \tag{7.6}$$

The detailed semantics of these operations are described in the respective detailed description of each individual operation.

7.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

7.1.1 C

NBLDequeCreateLF_HB Creates a new instance of a deque object using a lock-free implementation. See Section 7.3.1.

NBLDequeCreateLF_HU Creates a new instance of a deque object using a lock-free implementation. See Section 7.3.2.

NBLDequeCreateLF_LB Creates a new instance of a deque object using a lock-free implementation. See Section 7.3.3.

NBLDequeCreateLB Creates a new instance of a deque object using a lock-based implementation. See Section 7.3.4.

NBLDequePushLeft Puts a new item on top of the deque. See Section 7.4.1.

NBLDequePushRight Puts a new item on bottom of the deque. See Section 7.4.2.

NBLDequePopLeft Removes an item from top of the deque. See Section 7.4.3.

NBLDequePopRight Removes an item from bottom of the deque. See Section 7.4.4.

7.1.2 C++

```
namespace NBL {
  template <typename T> class Deque {
    // Constructors
    static Deque<T>* CreateLF_HB(int nrOfBlocks);
    static Deque<T>* CreateLF_HU(int nrOfBlocks);
    static Deque<T>* CreateLF_LB(int nrOfBlocks, int
        nrOfThreads);
    static Deque<T>* CreateLB();
    // Operations
    bool PushLeft(T* item);
    bool PushRight(T* item);
    T* PopLeft();
    T* PopRight();
  };
};
```

NBL::Deque::CreateLF_HB Creates a new instance of a deque object using a lock-free implementation. See Section 7.3.1.

NBL::Deque::CreateLF_HU Creates a new instance of a deque object using a lock-free implementation. See Section 7.3.2.

NBL::Deque::CreateLF_LB Creates a new instance of a deque object using a lock-free implementation. See Section 7.3.3.

NBL::Deque::CreateLB Creates a new instance of a deque object using a lock-based implementation. See Section 7.3.4.

NBL::Deque::PushLeft Puts a new item on top of the deque. See Section 7.4.1.

NBL::Deque::PushRight Puts a new item on bottom of the deque. See Section 7.4.2.

NBL::Deque::PopLeft Removes an item from top of the deque. See Section 7.4.3.

NBL::Deque::PopRight Removes an item from bottom of the deque. See Section 7.4.4.

7.2 Examples

7.2.1 C

```
NBLDequeRoot *deque;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free deque data object, with maximum space
      for appr. 100 items */
  deque = NBLDequeCreateLF_ST_GPST(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the deque object used */
  NBLDequeFree(deque);
  deque=NULL;
}

void Thread1()
{
  NBLDeque *handle;
  /* Get the local handle to the shared deque */
  handle=NBLDequeGetHandle(queue);

  /* Enqueue an item on the deque, i.e. putting items[3] on the
       deque */
  NBLDequePushLeft(handle,\&(items[3]));

  /* Free the local handle to the shared deque */
  NBLDequeFreeHandle(handle);
}

void Thread2()
{
  int *item;
  NBLDeque *handle;
  /* Get the local handle to the shared deque */
  handle=NBLDequeGetHandle(queue);

  /* Enqueue another item on the deque, i.e. putting items[2]
      on the deque */
  NBLDequePushRight(handle,\&(items[2]));

  /* Dequeueing an item from the deque i.e. we should get a
      pointer to items[3] or items[2] depending of which thread
      was the fastest */
```

```
  item = NBLDequePopLeft(handle);

  /* Free the local handle to the shared deque */
  NBLDequeFreeHandle(handle);
}

void Thread3()
...
```

7.2.2 C++

```
NBL::Deque<int> *deque;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free deque data object, with maximum space
      for appr. 100 items */
  deque = NBL::Deque<int>::CreateLF_HU(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the deque object used */
  delete deque;
  deque=NULL;
}

void Thread1()
{
  /* Enqueue an item on the deque, i.e. putting items[3] on the
       deque */
  deque->PushLeft(\&(items[3]));

}

void Thread2()
{
  int *item;

  /* Enqueue another item on the deque, i.e. putting items[2]
      on the deque */
  deque->PushRight(\&(items[2]));

  /* Dequeueing an item from the deque i.e. we should get a
      pointer to items[3] or items[2] depending of which thread
      was the fastest */
  item = deque->PopLeft();

}
```

```
void Thread3()
...
```

7.3 Creation

The following functions are used to create a new deque shared data structure.

7.3.1 NBLDequeCreateLF_HB

Creates a new instance of a deque object using a lock-free implementation.

The implementation allows concurrent operations that operate on separate items to perform in parallel, thus supporting high parallelism. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLDequeRoot* NBLDequeCreateLF_HB(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Deque<T>* NBL::Deque<T>::CreateLF_HB(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory blocks that can be used for the implementation of the deque. This measures approximately the number of items that can be stored into the deque: $nrOfBlocks = n + N^2 * 12 + N$, *where* n=nrOfItems, N=nrOfThreads

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a deque object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.3.2 NBLDequeCreateLF_HU

Creates a new instance of a deque object using a lock-free implementation.

The implementation allows concurrent operations that operate on separate items to perform in parallel, thus supporting high parallelism. The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLDequeRoot* NBLDequeCreateLF_HU(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::Deque<T>* NBL::Deque<T>::CreateLF_HU(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the deque. This measures approximately the number of items that can be stored into the deque: $nrOfBlocks \geq n + N * 11$, *where* n=nrOfItems, N=nrOfThreads

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a deque object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.3.3 NBLDequeCreateLF_LB

Creates a new instance of a deque object using a lock-free implementation.

The implementation does not allow concurrent operations to perform in parallel, thus supporting low parallelism. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLDequeRoot* NBLDequeCreateLF_LB(
  int nrOfBlocks,
  int nrOfThreads
);
```

Syntax C++

```
template <typename T> class
static NBL::Deque<T>* NBL::Deque<T>::CreateLF_LB(
  int nrOfBlocks = 1000,
  int nrOfThreads = 16
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory blocks that can be used for the implementation of the deque. This measures approximately the number of items that can be stored into the deque: $nrOfBlocks = n + N^2 * 6 + N$, *where* n=nrOfItems, N=nrOfThreads

nrOfThreads
: [in] The maximum number of concurrent threads that will access the structure.

Return Values

If successful, the function returns a pointer to a new instance of a deque object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.3.4 NBLDequeCreateLB

Creates a new instance of a deque object using a lock-based implementation.

Syntax C

```
NBLDequeRoot* NBLDequeCreateLB(
);
```

Syntax C++

```
template <typename T> class
static NBL::Deque<T>* NBL::Deque<T>::CreateLB(
);
```

Parameters

Return Values

If successful, the function returns a pointer to a new instance of a deque object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.4 Operations

7.4.1 NBLDequePushLeft

Puts a new item on top of the deque.

Syntax C

```
int NBLDequePushLeft(
  NBLDeque* handle,
  void* item
);
```

Syntax C++

```
template <typename T> class
bool NBL::Deque<T>::PushLeft(
  void* item
);
```

Parameters

handle
[in] A pointer to a local handle of a deque object instance.

item
[in] Pointer to the item to push on the deque. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the deque was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all enqueued items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.4.2 NBLDequePushRight

Puts a new item on the bottom of the deque.

Syntax C

```
int NBLDequePushRight(
  NBLDeque* handle,
  void* item
);
```

Syntax C++

```
template <typename T> class
bool NBL::Deque<T>::PushRight(
  void* item
);
```

Parameters

handle
: [in] A pointer to a local handle of a deque object instance.

item
: [in] Pointer to the item to push on the deque. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the deque was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all enqueued items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.4.3 NBLDequePopLeft

Removes an item from the top of the deque.

Syntax C

```
void* NBLDequePopLeft(
  NBLDeque* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::Deque<T>::PopLeft(
);
```

Parameters

handle
[in] A pointer to a local handle of a deque object instance.

Return Values

If successful, the function returns a pointer to the item that was on the top of the deque at some point in time and directly afterwards removed from the deque. Otherwise the deque was empty at some point in time, and the function returns NULL.

Remarks

The actual item is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

7.4.4 NBLDequePopRight

Removes an item from the bottom of the deque.

Syntax C

```
void* NBLDequePopRight(
  NBLDeque* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::Deque<T>::PopRight(
);
```

Parameters

handle
: [in] A pointer to a local handle of a deque object instance.

Return Values

If successful, the function returns a pointer to the item that was on the bottom of the deque at some point in time and directly afterwards removed from the deque. Otherwise the deque was empty at some point in time, and the function returns NULL.

Remarks

The actual item is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 8

Shared Priority Queue

The Priority Queue abstract data type is a collection of items which can efficiently support finding the item with the highest priority. Basic operations are *Insert* (add an item), *FindMin* (finds the item with minimum (or maximum) priority), and *DeleteMin* (removes the item with minimum (or maximum) priority). DeleteMin returns the item removed.

The implementations available of a shared priority queue are:

- A Lock-Free implementation offering expected logarithmic sequential time complexity, and bounded memory usage.
- A Lock-Free implementation offering expected logarithmic sequential time complexity, and unbounded memory usage.
- A Lock-Based implementation offering static structure, and deterministic logarithmic sequential time complexity.
- A Lock-Based implementation offering dynamic structure, and deterministic logarithmic sequential time complexity.
- A Lock-Based implementation offering expected logarithmic sequential time complexity.

All of the available implementations of a shared priority queue offers the same high-level semantics.

Definition 4 *We denote with L_t the abstract state of a priority queue at the time t. $L_t = [\langle p_1, v_1\rangle, \ldots, \langle p_n, v_n\rangle]$, where $|L_t| \geq 0$, is viewed as a list of pairs $\langle p, v\rangle$ consisting of a priority p and a corresponding value v. The operations that can be performed on the priority queue are* Insert, FindMin, *and* DeleteMin. *The time t_1 is defined as the time just before the operation of interest takes place, and the time t_2 is defined as the time just after the same operation takes place. In the following expressions that define the high-level semantics of our operations, the syntax is S_1 :*

O_1, S_2, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:

$$L_{t_1} = L_1 + [\langle p_1, v_1\rangle] + [\langle p_3, v_3\rangle] + L_2 \wedge p_1 <= p_2 \wedge p_2 < p_3 : \mathbf{Insert}(\langle \mathbf{p_2}, \mathbf{v_2}\rangle) = \mathbf{true}, L_{t_2} = L_1 + [\langle p_1, v_1\rangle] + [\langle p_2, v_2\rangle] + [\langle p_3, v_3\rangle] + L_2$$

$$L_{t_1} = [\langle p_1, v_1\rangle] + L_1 : \mathbf{FindMin}() = \langle \mathbf{p_1}, \mathbf{v_1}\rangle,\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}}$$

$$L_{t_1} = \emptyset : \mathbf{FindMin}() = \bot,\ \mathbf{L_{t_2}} = \mathbf{L_1}$$

$$L_{t_1} = [\langle p_1, v_1\rangle] + L_1 : \mathbf{DeleteMin}() = \langle \mathbf{p_1}, \mathbf{v_1}\rangle,\ \mathbf{L_{t_2}} = \mathbf{L_1}$$

$$L_{t_1} = \emptyset : \mathbf{DeleteMin}() = \bot,\ \mathbf{L_{t_2}} = \mathbf{L_1}$$

The detailed semantics of these operations are described in the respective detailed description of each individual operation.

8.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

8.1.1 C

NBLPQueueCreateLF_EB Creates a new instance of a priority queue object using a lock-free implementation. See Section 8.3.1

NBLPQueueCreateLF_EU Creates a new instance of a priority queue object using a lock-free implementation. See Section 8.3.2

NBLPQueueCreateLB_SD Creates a new instance of a priority queue object using a lock-based implementation. See Section 8.3.3

NBLPQueueCreateLB_DD Creates a new instance of a priority queue object using a lock-based implementation. See Section 8.3.4

NBLPQueueCreateLB_E Creates a new instance of a priority queue object using a lock-based implementation. See Section 8.3.5

NBLPQueueInsert Inserts a new item. See Section 8.4.1.

NBLPQueueDeleteMin Removes the item with the lowest priority. See Section 8.4.2.

NBLPQueueFindMin Finds the item with the lowest priority. See Section 8.4.3.

8.1.2 C++

```
namespace NBL {
  template <typename T, typename P = int> class PQueue {
    // Constructors
    static PQueue<T,P>* CreateLF_EB(int nrOfBlocks, int
        avgNodes);
    static PQueue<T,P>* CreateLF_EU(int nrOfBlocks, int
        avgNodes);
    static PQueue<T,P>* CreateLB_E(int avgNodes);
    static PQueue<T,P>* CreateLB_SD(int nrOfBlocks, int
        nrOfThreads);
    static PQueue<T,P>* CreateLB_DD();
    // Operations
    bool Insert(P* priority, T *item);
    T* FindMin(P** priority);
    T* DeleteMin(P** priority);
  };

  template <typename T, int> class PQueue \{
    // Constructors
    static PQueue<T,int>* CreateLF_EB(int nrOfBlocks, int
        avgNodes);
    static PQueue<T,int>* CreateLF_EU(int nrOfBlocks, int
        avgNodes);
    static PQueue<T,int>* CreateLB_E(int avgNodes);
    static PQueue<T,int>* CreateLB_SD(int nrOfBlocks, int
        nrOfThreads);
    static PQueue<T,int>* CreateLB_DD();
    // Operations
    bool Insert(int priority, T *item);
    T* FindMin(int * priority);
    T* DeleteMin(int * priority);
  };
};

// Definied elsewhere for typename P
bool operator < (const P &t1, const P &t2);
bool operator == (const P &t1, const P &t2);
```

NBL::PQueue::CreateLF_EB Creates a new instance of a priority queue object using a lock-free implementation. See Section 8.3.1

NBL::PQueue::CreateLF_EU Creates a new instance of a priority queue object using a lock-free implementation. See Section 8.3.2

NBL::PQueue::CreateLB_SD Creates a new instance of a priority queue object using a lock-based implementation. See Section 8.3.3

NBL::PQueue::CreateLB_DD Creates a new instance of a priority queue object using a lock-based implementation. See Section 8.3.4

NBL::PQueue::CreateLB_E Creates a new instance of a priority queue object using a lock-based implementation. See Section 8.3.5

NBL::PQueue::Insert Inserts a new item. See Section 8.4.1.

NBL::PQueue::DeleteMin Removes the item with the lowest priority. See Section 8.4.2.

NBL::PQueue::FindMin Finds the item with the lowest priority. See Section 8.4.3.

8.2 Examples

8.2.1 C

```
NBLPQueueRoot *pqueue;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free priority queue object, with maximum
     space for appr. 100 items */
  pqueue = NBLPQueueCreateLF_EB(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the list object used */
  NBLPQueueFree(pqueue);
  pqueue=NULL;
}

void Thread1()
{
  NBLPQueue *handle;
  /* Get the local handle to the shared priority queue */
  handle=NBLPQueueGetHandle(pqueue);

  /* Insert an item in the queue, i.e. putting items[3] on the
     queue */
```

```
  NBLPQueueInsert(handle,&(items[3]),4);

  /* Free the local handle to the shared priority queue */
  NBLPQueueFreeHandle(handle);
}

void Thread2()
{
  int *item;
  NBLPQueue *handle;
  /* Get the local handle to the shared priority queue */
  handle=NBLPQueueGetHandle(pqueue);

  /* Enlist another item on the queue, i.e. putting items[2] on
      the queue */
  NBLPQueueInsert(handle,&(items[2]),3);

  /* Deleting an item from the queue i.e. we should get a
     pointer to items[3] or items[2] depending of which thread
     was the fastest */
  item = NBLPQueueDeleteMin(handle);

  /* Free the local handle to the shared priority queue */
  NBLPQueueFreeHandle(handle);
}

void Thread3()
...
```

8.2.2 C++

```
NBL::PQueue<int,int> *pqueue;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free priority queue object, with maximum
     space for appr. 100 items */
  pqueue = NBL::PQueue<int,int>::CreateLF_EB(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the list object used */
  delete pqueue;
  pqueue=NULL;
}

void Thread1()
{
```

```
  /* Insert an item in the queue, i.e. putting items[3] on the
      queue */
  pqueue->Insert(&(items[3]),4);

}

void Thread2()
{
  int *item;

  /* Enlist another item on the queue, i.e. putting items[2] on
       the queue */
  pqueue->Insert(&(items[2]),3);

  /* Deleting an item from the queue i.e. we should get a
      pointer to items[3] or items[2] depending of which thread
      was the fastest */
  item = pqueue->DeleteMin();

}

void Thread3()
...
```

8.3 Creation

The following functions are used to create a new priority queue shared data structure.

The implementation uses randomization and permits an expected logarithmic time complexity for insert operations that execute sequentially. The amount of memory occupied at any moment in time for storing the items is bounded.

8.3.1 NBLPQueueCreateLF_EB

Creates a new instance of a priority queue object using a lock-free implementation.

Syntax C

```
NBLPQueueRoot* NBLPQueueCreateLF_EB(
  int nrOfBlocks,
  int avgNodes
);
```

Syntax C++

```
template <typename T, typename P = int> class
static NBL::PQueue<T,P>* NBL::PQueue<T,P>::CreateLF_EB(
  int nrOfBlocks = 1000,
  int avgNodes = 500
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory blocks that can be used for the implementation of the priority queue. This measures approximately the number of items that can be stored into the priority queue:
$nrOfBlocks = n+N^2*(10+2*maxLevel)+N$, *where* n=nrOfItems, N=nrOfThreads

avgNodes
: [in] The expected average number of items in the structure.

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

PARAM_PRI_COMPARE_FN

For detailed parameter descriptions see Sections 2.4.3 and 2.4.4.

Return Values

If successful, the function returns a pointer to a new instance of a priority queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.3.2 NBLPQueueCreateLF_EU

Creates a new instance of a priority queue object using a lock-free implementation.

The implementation uses randomization and permits an expected logarithmic time complexity for insert operations that execute sequentially. The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLPQueueRoot* NBLPQueueCreateLF_EU(
  int nrOfBlocks,
  int avgNodes
);
```

Syntax C++

```
template <typename T, typename P = int> class
static NBL::PQueue<T,P>* NBL::PQueue<T,P>::CreateLF_EU(
  int nrOfBlocks = 1000,
  int avgNodes = 500
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory cells that can be used for the implementation of the priority queue. This measures approximately the number of actual items that can be stored into the priority queue.

avgNodes
: [in] The expected average number of items in the structure.

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

PARAM_PRI_COMPARE_FN

For detailed parameter descriptions see Sections 2.4.3 and 2.4.4.

Return Values

If successful, the function returns a pointer to a new instance of a priority queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.3.3 NBLPQueueCreateLB_SD

Creates a new instance of a priority queue object using a lock-based implementation.

The items are stored in a static structure. The implementation permits a deterministic logarithmic time complexity for insert operations that execute sequentially.

Syntax C

```
NBLPQueueRoot* NBLPQueueCreateLB_SD(
  int nrOfBlocks,
  int nrOfThreads
);
```

Syntax C++

```
template <typename T, typename P = int> class
static NBL::PQueue<T,P>* NBL::PQueue<T,P>::CreateLB_SD(
  int nrOfBlocks = 1000,
  int nrOfThreads = 16
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory cells that can be used for the implementation of the priority queue. This measures approximately the number of actual items that can be stored into the structure.

nrOfThreads
: [in] The maximum number of concurrent threads that will access the structure.

On-line Parameters

PARAM_PRI_COMPARE_FN

For detailed parameter descriptions see Section 2.4.4.

Return Values

If successful, the function returns a pointer to a new instance of a priority queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.3.4 NBLPQueueCreateLB_DD

Creates a new instance of a priority queue object using a lock-based implementation.

The items are stored in a dynamic structure. The implementation permits a deterministic logarithmic time complexity for insert operations that execute sequentially.

Syntax C

```
NBLPQueueRoot* NBLPQueueCreateLB_DD(
);
```

Syntax C++

```
template <typename T, typename P = int> class
static NBL::PQueue<T,P>* NBL::PQueue<T,P>::CreateLB_DD(
);
```

Parameters

On-line Parameters

PARAM_PRI_COMPARE_FN

For detailed parameter descriptions see Section 2.4.4.

Return Values

If successful, the function returns a pointer to a new instance of a priority queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.3.5 NBLPQueueCreateLB_E

Creates a new instance of a priority queue object using a lock-based implementation.

The implementation uses randomization and permits an expected logarithmic time complexity for insert operations that execute sequentially.

Syntax C

```
NBLPQueueRoot* NBLPQueueCreateLB_E(
  int avgNodes
);
```

Syntax C++

```
template <typename T, typename P = int> class
static NBL::PQueue<T,P>* NBL::PQueue<T,P>::CreateLB_E(
  int avgNodes = 500
);
```

Parameters

avgNodes
[in] The expected average number of items in the structure.

On-line Parameters

PARAM_PRI_COMPARE_FN

For detailed parameter descriptions see Section 2.4.4.

Return Values

If successful, the function returns a pointer to a new instance of a priority queue object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.4 Operations

8.4.1 NBLPQueueInsert

Inserts a new item.

Syntax C

```
int NBLPQueueInsert(
  NBLPQueue* handle,
  void* priority,
  void* item
);
```

Syntax C++

```
template <typename T, int> class
bool NBL::PQueue<T,int>::Insert(
  int priority,
  T* item
);

template <typename T, typename P> class
bool NBL::PQueue<T,P>::Insert(
  P* priority,
  T* item
);
```

Parameters

handle
: [in] A pointer to a local handle of a priority queue object instance.

priority
: [in] Pointer to the priority of the item to insert in the priority queue. This should point to a memory resident structure which stays valid after the function call.

item
: [in] Pointer to the item to insert in the priority queue. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the priority queue was full (i.e. out of memory), and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all inserted items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.4.2 NBLPQueueDeleteMin

Removes the item with the lowest priority.

Syntax C

```
void* NBLPQueueDeleteMin(
  NBLPQueue* handle,
  void **priority
);
```

Syntax C++

```
template <typename T, int> class
T* NBL::PQueue<T,int>::DeleteMin(
  int* priority = NULL
);

template <typename T, typename P> class
T* NBL::PQueue<T,P>::DeleteMin(
  P** priority = NULL
);
```

Parameters

handle
: [in] A pointer to a local handle of a priority queue object instance.

priority
: [out] Optional. NULL or a pointer to valid memory where a pointer to the priority of the deleted item will be stored.

Return Values

If successful, the function returns a pointer to the item that was removed. Otherwise the priority queue was empty and the function returns NULL.

Remarks

The actual item is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

8.4.3 NBLPQueueFindMin

Finds the item with the lowest priority.

Syntax C

```
void* NBLPQueueFindMin(
  NBLPQueue* handle,
  void** priority
);
```

Syntax C++

```
template <typename T, int> class
T* NBL::PQueue<T,int>::FindMin(
  int* priority = NULL
);

template <typename T, typename P> class
T* NBL::PQueue<T,P>::FindMin(
  P** priority = NULL
);
```

Parameters

handle
: [in] A pointer to a local handle of a priority queue object instance.

priority
: [out] Optional. NULL or a pointer to valid memory where a pointer to the priority of the item will be stored.

Return Values

If successful, the function returns a pointer to the item. Otherwise the priority queue was empty and the function returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 9

Shared Dictionary

The Dictionary abstract data type is a collection of items where each item is associated with a key. The data structure should efficiently support finding the item associated with the specific key. Basic operations are *Insert* (add an item associated with a key), *Find* (finds the item associated with a certain key), and *Delete* (removes the item associated with a certain key). Delete returns the item removed. In a concurrent environment an additional basic operation is *Update* (re-assign the association of a key with a certain item).

The implementations available of a shared dictionary are:

- A Lock-Free implementation offering expected logarithmic sequential time complexity, and bounded memory usage.
- A Lock-Free implementation offering expected logarithmic sequential time complexity, and unbounded memory usage.
- A Lock-Free implementation offering linear sequential time complexity, and bounded memory usage.
- A Lock-Based implementation offering expected logarithmic sequential time complexity.

All of the available implementations of a shared priority queue offers the same high-level semantics.

Definition 5 *We denote with L_t the abstract state of a dictionary at the time t. $L_t = \{\langle k_1, v_1\rangle, \ldots, \langle k_n, v_n\rangle\}$, where $|L_t| \geq 0$, is viewed as a set of pairs $\langle k, v\rangle$ consisting of a unique key k and a corresponding value v. The operations that can be performed on the dictionary are* Insert, Update, Find, *and* Delete. *The time t_1 is defined as the time just before the operation of interest takes place, and the time t_2 is defined as the time just after the same operation takes place. In the following expressions that define the high-level semantics of our operations, the syntax is S_1 :*

O_1, S_2, where S_1 is the conditional state before the operation O_1, and S_2 is the resulting state after performing the corresponding operation:

$$\langle k_1, \_\rangle \notin L_{t_1} : \mathbf{Insert}(\langle \mathbf{k_1}, \mathbf{v_1}\rangle) = \mathbf{true}, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \cup \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{9.1}$$

$$\langle k_1, \_\rangle \in L_{t_1} : \mathbf{Insert}(\langle \mathbf{k_1}, \mathbf{v_1}\rangle) = \bot, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \tag{9.2}$$

$$\langle k_1, \_\rangle \notin L_{t_1} : \mathbf{Update}(\langle \mathbf{k_1}, \mathbf{v_1}\rangle) = \mathbf{true}, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \cup \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{9.3}$$

$$\begin{aligned} \langle k_1, v_{1_1}\rangle \in L_{t_1} : \mathbf{Update}(\langle \mathbf{k_1}, \mathbf{v_{1_2}}\rangle) = \mathbf{v_{1_1}}, \\ \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_{1_1}}\rangle\} \cup \{\langle \mathbf{k_1}, \mathbf{v_{1_2}}\rangle\} \end{aligned} \tag{9.4}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{Find}(\mathbf{k_1}) = \mathbf{v_1} \tag{9.5}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{Find}(\mathbf{k_1}) = \bot \tag{9.6}$$

$$\langle k_1, v_1\rangle \in L_{t_1} : \mathbf{Delete}(\mathbf{k_1}) = \mathbf{v_1}, \mathbf{L_{t_2}} = \mathbf{L_{t_1}} \setminus \{\langle \mathbf{k_1}, \mathbf{v_1}\rangle\} \tag{9.7}$$

$$\langle k_1, v_1\rangle \notin L_{t_1} : \mathbf{Delete}(\mathbf{k_1}) = \bot \tag{9.8}$$

9.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

9.1.1 C

NBLDictionaryCreateLF_EB Creates a new instance of a dictionary object using a lock-free implementation. See Section 9.3.1.

NBLDictionaryCreateLF_EU Creates a new instance of a dictionary object using a lock-free implementation. See Section 9.3.2.

NBLDictionaryCreateLF_LB Creates a new instance of a dictionary object using a lock-free implementation. See Section 9.3.3.

NBLDictionaryCreateLB_E Creates a new instance of a dictionary object using a lock-based implementation. See Section 9.3.4.

NBLDictionaryInsert Inserts a new association. See Section 9.4.1

NBLDictionaryUpdate Updates an existing association. See Section 9.4.2

NBLDictionaryDelete Deletes an association. See Section 9.4.3

NBLDictionaryFind Finds the value associated with a certain key. See Section 9.4.4.

9.1.2 C++

```
namespace NBL {
  template <typename T, typename K = int> class Dictionary {
    // Constructors
    static Dictionary<T,K>* CreateLF_EB(int nrOfBlocks, int
        avgNodes);
    static Dictionary<T,K>* CreateLF_EU(int nrOfBlocks, int
        avgNodes);
    static Dictionary<T,K>* CreateLF_LB(int nrOfBlocks);
    static Dictionary<T,K>* CreateLB_E(int avgNodes);
    // Operations
    bool Insert(K* key, T* item);
    bool Update(K* key, T* item, T** old);
    T *Delete(K* key);
    T *Find(K* key);
    // Auxiliary functions
    void SetValueMemoryHandler(Memory<T> *memory);
  };

  template <typename T, int> class Dictionary {
    // Constructors
    static Dictionary<T,int>* CreateLF_EB(int nrOfBlocks, int
        avgNodes);
    static Dictionary<T,int>* CreateLF_EU(int nrOfBlocks, int
        avgNodes);
    static Dictionary<T,int>* CreateLF_LB(int nrOfBlocks);
    static Dictionary<T,int>* CreateLB_E(int avgNodes);
    // Operations
    bool Insert(int key, T* item);
    bool Update(int key, T* item, T** old);
    T *Delete(int key);
    T *Find(int key);
    // Auxiliary functions
    void SetValueMemoryHandler(Memory<T> *memory);
  };
};

// Definied elsewhere for typename K
```

```
bool operator < (const K &t1, const K &t2);
bool operator == (const K &t1, const K &t2);
```

NBL::Dictionary::CreateLF_EB Creates a new instance of a dictionary object using a lock-free implementation. See Section 9.3.1.

NBL::Dictionary::CreateLF_EU Creates a new instance of a dictionary object using a lock-free implementation. See Section 9.3.2.

NBL::Dictionary::CreateLF_LB Creates a new instance of a dictionary object using a lock-free implementation. See Section 9.3.3.

NBL::Dictionary::CreateLB_E Creates a new instance of a dictionary object using a lock-based implementation. See Section 9.3.4.

NBL::Dictionary::Insert Inserts a new association. See Section 9.4.1

NBL::Dictionary::Update Updates an existing association. See Section 9.4.2

NBL::Dictionary::Delete Deletes an association. See Section 9.4.3

NBL::Dictionary::Find Finds the value associated with a certain key. See Section 9.4.4.

NBL::Dictionary::SetValueMemoryHandler Sets the custom value memory manager. See Section 9.4.5

9.2 Examples

9.2.1 C

```
NBLDictionary *dict;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free dictionary object, with maximum space
     for appr. 100 items */
  dict = NBLDictionaryCreateLF_EB(100,100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the dictionary object used */
  NBLDictionaryFree(dict);
  dict=NULL;
}
```

```
void Thread1()
{
  NBLDictionary *handle;
  /* Get the local handle to the shared dictionary */
  handle=NBLDictionaryGetHandle(dict);

  /* Insert an item in the dictionary, i.e. associating items
      [3] with
  the key 1 in the list */
  NBLDictionaryInsert(handle,1,&(items[3]));

  /* Free the local handle to the shared dictionary */
  NBLDictionaryFreeHandle(handle);
}

void Thread2()
{
  int *item;
  NBLDictionary *handle;
  /* Get the local handle to the shared dictionary */
  handle=NBLDictionaryGetHandle(dict);

  /* Insert another item on the dictionary, i.e. associating
      items[2]
  with the dictionary */
  NBLDictionaryInsert(handle,2,&(items[2]));

  /* Deleting an association from the dictionary i.e. we should
      get
  a pointer to items[3] if Thread1 has been run so far */
  item = NBLDictionaryDelete(handle,1);

  /* Free the local handle to the shared dictionary */
  NBLDictionaryFreeHandle(handle);
}

void Thread3()
...
```

9.2.2 C++

```
NBL::Dictionary<int,int> *dict;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free dictionary object, with maximum space
      for appr. 100 items */
  dict = NBL::Dictionary<int,int>::CreateLF_EB(100,100);
```

```
  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the dictionary object used */
  delete dict;
  dict=NULL;
}

void Thread1()
{
  /* Insert an item in the dictionary, i.e. associating items
      [3] with
  the key 1 in the list */
  dict->Insert(1,&(items[3]));

}

void Thread2()
{
  int *item;

  /* Insert another item on the dictionary, i.e. associating
      items[2]
  with the dictionary */
  dict->Insert(2,&(items[2]));

  /* Deleting an association from the dictionary i.e. we should
       get
  a pointer to items[3] if Thread1 has been run so far */
  item = dict->Delete(1);

}

void Thread3()
...
```

9.3 Creation

The following functions are used to create a new dictionary shared data structure.

9.3.1 NBLDictionaryCreateLF_EB

Creates a new instance of a dictionary object using a lock-free implementation.

The implementation uses randomization and permits an expected logarithmic time complexity for insert operations that execute sequentially. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLDictionaryRoot* NBLDictionaryCreateLF_EB(
  int nrOfBlocks,
  int avgNodes
);
```

Syntax C++

```
template <typename T, typename K = int> class
static NBL::Dictionary<T,K>* NBL::Dictionary<T,K>::CreateLF_EB(
  int nrOfBlocks = 1000,
  int avgNodes = 500
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory cells that can be used for the implementation of the dictionary. This measures approximately the number of actual items that can be stored into the dictionary.

avgNodes
: [in] The expected average number of items in the structure.

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

PARAM_KEY_COMPARE_FN

PARAM_VALUE_MEMORY_HANDLE

PARAM_VALUE_MEMORY_ROOT

For detailed parameter descriptions see Sections 2.4.3 and 2.4.5 and 2.4.2.

Return Values

If successful, the function returns a pointer to a new instance of a dictionary object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.3.2 NBLDictionaryCreateLF_EU

Creates a new instance of a dictionary object using a lock-free implementation.

The implementation uses randomization and permits an expected logarithmic time complexity for insert operations that execute sequentially. The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLDictionaryRoot* NBLDictionaryCreateLF_EU(
  int nrOfBlocks,
  int avgNodes
);
```

Syntax C++

```
template <typename T, typename K = int> class
static NBL::Dictionary<T,K>* NBL::Dictionary<T,K>::CreateLF_EU(
  int nrOfBlocks = 1000,
  int avgNodes = 500
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory cells that can be used for the implementation of the dictionary. This measures approximately the number of actual items that can be stored into the dictionary.

avgNodes
[in] The expected average number of items in the structure.

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

PARAM_KEY_COMPARE_FN

PARAM_VALUE_MEMORY_HANDLE
PARAM_VALUE_MEMORY_ROOT

For detailed parameter descriptions see Sections 2.4.3 and 2.4.5 and 2.4.2.

Return Values

If successful, the function returns a pointer to a new instance of a dictionary object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.3.3 NBLDictionaryCreateLF_LB

Creates a new instance of a dictionary object using a lock-free implementation.

The implementation permits a linear time complexity for insert operations that execute sequentially. The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLDictionaryRoot* NBLDictionaryCreateLF_LB(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T, typename K = int> class
static NBL::Dictionary<T,K>* NBL::Dictionary<T,K>::CreateLF_LB(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
: [in] The maximum number of memory cells that can be used for the implementation of the dictionary. This measures approximately the number of actual items that can be stored into the dictionary.

On-line Parameters

PARAM_KEY_COMPARE_FN

PARAM_VALUE_MEMORY_HANDLE
PARAM_VALUE_MEMORY_ROOT

For detailed parameter descriptions see Sections 2.4.5 and 2.4.2.

Return Values

If successful, the function returns a pointer to a new instance of a dictionary object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.3.4 NBLDictionaryCreateLB_E

Creates a new instance of a dictionary object using a lock-based implementation.

The implementation uses randomization and permits an expected logarithmic time complexity for insert operations that execute sequentially.

Syntax C

```
NBLDictionaryRoot* NBLDictionaryCreateLB_E(
  int avgNodes
);
```

Syntax C++

```
template <typename T, typename K = int> class
static NBL::Dictionary<T,K>* NBL::Dictionary<T,K>::CreateLB_E(
  int avgNodes = 500
);
```

Parameters

avgNodes
[in] The expected average number of items in the structure.

On-line Parameters

PARAM_KEY_COMPARE_FN

PARAM_VALUE_MEMORY_HANDLE
PARAM_VALUE_MEMORY_ROOT

For detailed parameter descriptions see Sections 2.4.5 and 2.4.2.

Return Values

If successful, the function returns a pointer to a new instance of a dictionary object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.4 Operations

9.4.1 NBLDictionaryInsert

Inserts a new association.

Syntax C

```
int NBLDictionaryInsert(
  NBLDictionary* handle,
  void* key,
  void* value
);
```

Syntax C++

```
template <typename T, int> class
bool NBL::Dictionary<T,int>::Insert(
  int key,
  K* value
);

template <typename T, typename K> class
bool NBL::Dictionary<T,K>::Insert(
  K* key,
  T* value
);
```

Parameters

handle
: [in] A pointer to a local handle of a dictionary object instance.

key
: [in] Pointer to the key to insert in the dictionary. This should point to a memory resident structure which stays valid after the function call.

value
: [in] Pointer to the value to insert in the dictionary. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the dictionary already contained an association with the given key or the dictionary was full (i.e. out of memory), and the function returns false.

Remarks

Only the pointers to the new value objects are copied, which means that the user has to provide some kind of memory management system to be able to keep all inserted value objects valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.4.2 NBLDictionaryUpdate

Updates an existing association, or inserts a new if not already present.

Syntax C

```
int NBLDictionaryUpdate(
  NBLDictionary* handle,
  void* key,
  void* value,
  void** old
);
```

Syntax C++

```
template <typename T, int> class
bool NBL::Dictionary<T,int>::Update(
  int key,
  T* value,
  T** old = NULL
);

template <typename T, typename K> class
bool NBL::Dictionary<T,K>::Update(
  K* key,
  T* value,
  T** old = NULL
);
```

Parameters

handle
: [in] A pointer to a local handle of a dictionary object instance.

key
: [in] Pointer to the key which association should be updated in the dictionary. This should point to a memory resident structure which stays valid after the function call.

value
: [in] Pointer to the value to associate with the given key in the dictionary. This should point to a memory resident structure which stays valid after the function call.

old
: [out] Optional. NULL or a pointer to valid memory where a pointer to the old value that was associated with the given key in the dictionary will be stored.

Return Values

If successful, the function returns true. Otherwise the dictionary was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new value objects are copied, which means that the user has to provide some kind of memory management system to be able to keep all inserted value objects valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.4.3 NBLDictionaryDelete

Removes an association with a certain key.

Syntax C

```
void* NBLDictionaryDelete(
  NBLDictionary* handle,
  void *key
);
```

Syntax C++

```
template <typename T, int> class
T* NBL::Dictionary<T,int>::Delete(
  int key
);

template <typename T, typename K> class
T* NBL::Dictionary<T,K>::Delete(
  K *key
);
```

Parameters

handle
: [in] A pointer to a local handle of a dictionary object instance.

key
: [in] Pointer to the key which association should be deleted in the dictionary.

Return Values

If successful, the function returns a pointer to the value that was associated with the the given key in the dictionary. Otherwise the association was already deleted and the function returns NULL.

Remarks

The actual value object is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.4.4 NBLDictionaryFind

Returns the value that is associated with a certain key.

Syntax C

```
void* NBLDictionaryFind(
  NBLDictionary* handle,
  void* key
);
```

Syntax C++

```
template <typename T, int> class
T* NBL::Dictionary<T,int>::Find(
  int key
);

template <typename T, typename K> class
T* NBL::Dictionary<T,K>::Find(
  K* key
);
```

Parameters

handle
: [in] A pointer to a local handle of a dictionary object instance.

key
: [in] Pointer to the key which association should be found.

Return Values

If successful, the function returns a pointer to the value that is associated with the given key in the dictionary. Otherwise there was no association in the dicionary, and the function returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

9.4.5 NBL::Dictionary::SetValueMemoryHandler

Sets the custom value memory manager.

Syntax C++

```
template <typename T, int> class
void NBL::Dictionary<T,int>::SetValueMemoryHandler (
  Memory<T>* memory
);

template <typename T, typename K> class
void NBL::Dictionary<T,K>::SetValueMemoryHandler (
  Memory<T>* memory
);
```

Parameters

memory
[in] Pointer to the memory manager object to be used for values.

Return Values

None.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 10

Shared List

The List abstract data type is a collection of items where two items are related only with respect to their relative position to each other. The data structure should efficiently support traversals among the items. In a concurrent environment, traversals to absolute index positions are not feasible. Consequently, traversals are only supported relatively to a current position. The current position is maintained by the cursor concept, where each handle (i.e. thread or process) maintains one independent cursor position. The first and last cursor positions do not refer to real items, but are instead used as end markers, i.e. before the first item or after the last item. Basic operations are *InsertAfter* (add a new item after the current), *Delete* (remove the current item), *Read* (inspect the current item), *Next* (traverse to the item after the current), *First* (traverse to the position before the first item). Additional operations are *InsertBefore* (add a new item before the current), *Previous* (traverse to the item before the current), and *Last* (traverse to the position after the last item).

The list is normally implemented using a linked list data structure. This can be either singly or doubly linked, i.e. either each item knows only about its successor, or it knows about both its successor and its predecessor. Even though all implementations support traversals in both directions, those based on a doubly linked list facilitates significantly more efficient backward traversals.

The implementations available of a shared list are:

- A Lock-Free implementation offering singly linked list characteristics, and unbounded memory usage.

- A Lock-Based implementation offering singly linked list characteristics.

- A Lock-Free implementation offering doubly linked list characteristics, and bounded memory usage.

- A Lock-Free implementation offering doubly linked list characteristics, and unbounded memory usage.
- A Lock-Based implementation offering doubly linked list characteristics.

10.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

10.1.1 C

NBLListCreateLF_SU Creates a new instance of a singly linked list object using a lock-free implementation. See Section 10.3.1.

NBLListCreateLB_S Creates a new instance of a singly linked list object using a lock-based implementation. See Section 10.3.2.

NBLListCreateLF_DB Creates a new instance of a doubly linked list object using a lock-free implementation. See Section 10.3.3.

NBLListCreateLF_DU Creates a new instance of a doubly linked list object using a lock-free implementation. See Section 10.3.4.

NBLListCreateLB_D Creates a new instance of a doubly linked list object using a lock-based implementation. See Section 10.3.5.

NBLListInsertBefore Inserts a new element directly before the current position. See Section 10.4.1.

NBLListInsertAfter Inserts a new element directly after the current position. See Section 10.4.2.

NBLListDelete Deletes the element at the current position. See Section 10.4.3.

NBLListRead Reads the element at the current position. See Section 10.4.4.

NBLListFirst Sets the cursor position to point directly before the first element. See Section 10.4.5.

NBLListLast Sets the cursor position to point directly after the last element. See Section 10.4.6.

NBLListNext Traverses the cursor position one step forwards. See Section 10.4.7.

NBLListPrevious Traverses the cursor position one step backwards. See Section 10.4.8.

10.1.2 C++

```
namespace NBL {
  template <typename T> class List {
    // Constructors
    static List<T> *CreateLF_SU(int nrOfBlocks);
    static List<T> *CreateLB_S();
    static List<T> *CreateLF_DB(int nrOfBlocks);
    static List<T> *CreateLF_DU(int nrOfBlocks);
    static List<T> *CreateLB_D();
    // Operations
    bool InsertBefore(T *item);
    bool InsertAfter(T *item);
    T *Delete();
    T *Read();
    void First();
    void Last();
    bool Next();
    bool Previous();
  };
};
```

NBL::List::CreateLF_SU Creates a new instance of a singly linked list object using a lock-free implementation. See Section 10.3.1.

NBL::List::CreateLB_S Creates a new instance of a singly linked list object using a lock-based implementation. See Section 10.3.2.

NBL::List::CreateLF_DB Creates a new instance of a doubly linked list object using a lock-free implementation. See Section 10.3.3.

NBL::List::CreateLF_DU Creates a new instance of a doubly linked list object using a lock-free implementation. See Section 10.3.4.

NBL::List::CreateLB_D Creates a new instance of a doubly linked list object using a lock-based implementation. See Section 10.3.5.

NBL::List::InsertBefore Inserts a new element directly before the current position. See Section 10.4.1.

NBL::List::InsertAfter Inserts a new element directly after the current position. See Section 10.4.2.

NBL::List::Delete Deletes the element at the current position. See Section 10.4.3.

NBL::List::Read Reads the element at the current position. See Section 10.4.4.

NBL::List::First Sets the cursor position to point directly before the first element. See Section 10.4.5.

NBL::List::Last Sets the cursor position to point directly after the last element. See Section 10.4.6.

NBL::List::Next Traverses the cursor position one step forwards. See Section 10.4.7.

NBL::List::Previous Traverses the cursor position one step backwards. See Section 10.4.8.

10.2 Examples

10.2.1 C

```
NBLListRoot *list;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free linked list data object, with maximum
      space for appr. 100 items */
  list = NBLListCreateLF_DB(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the list object used */
  NBLListFree(list);
  list=NULL;
}

void Thread1()
{
  NBLList *handle;
  /* Get the local handle to the shared linked list */
  handle=NBLListGetHandle(list);

  /* Insert an item in the list, i.e. putting items[3] on the
      list */
  NBLListInsertAfter(handle,&(items[3]));

  /* Free the local handle to the shared linked list */
  NBLListFreeHandle(handle);
}

void Thread2()
{
  int *item;
  NBLList *handle;
```

```
  /* Get the local handle to the shared linked list */
  handle=NBLListGetHandle(list);

  /* Enlist another item on the list, i.e. putting items[2] on
      the list */
  NBLListInsertAfter(handle,&(items[2]));

  /* Deleting an item from the list i.e. we should get a
      pointer to items[3] or items[2] depending of which thread
      was the fastest */
  item = NBLListDelete(handle);

  /* Free the local handle to the shared linked list */
  NBLListFreeHandle(handle);
}

void Thread3()
...
```

10.2.2 C++

```
NBL::List<int> *list;
int items[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a lock-free linked list data object, with maximum
      space for appr. 100 items */
  list = NBL::List<int>::CreateLF_DB(100);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the list object used */
  delete list;
  list=NULL;
}

void Thread1()
{
  /* Insert an item in the list, i.e. putting items[3] on the
      list */
  list->InsertAfter(&(items[3]));

}

void Thread2()
{
  int *item;
```

```
  /* Enlist another item on the list, i.e. putting items[2] on
     the list */
  list->InsertAfter(&(items[2]));

  /* Deleting an item from the list i.e. we should get a
     pointer to items[3] or items[2] depending of which thread
     was the fastest */
  item = list->Delete();

}

void Thread3()
...
```

10.3 Creation

The following functions are used to create a new linked list shared data structure.

10.3.1 NBLListCreateLF_SU

Creates a new instance of a singly linked list object using a lock-free implementation.

The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLListRoot* NBLListCreateLF_SU(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::List<T>* NBL::List<T>::CreateLF_SU(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the list. This measures approximately the number of items that can be stored into the list: $nrOfBlocks \geq n + N * 5$, *where* n=nrOfItems, N=nrOfThreads

Return Values

If successful, the function returns a pointer to a new instance of a linked list object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.3.2 NBLListCreateLB_S

Creates a new instance of a singly linked list object using a lock-based implementation.

Syntax C

```
NBLListRoot* NBLListCreateLB_S(
);
```

Syntax C++

```
template <typename T> class
static NBL::List<T>* NBL::List<T>::CreateLB_S(
);
```

Parameters

Return Values

If successful, the function returns a pointer to a new instance of a linked list object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.3.3 NBLListCreateLF_DB

Creates a new instance of a doubly linked list object using a lock-free implementation.

The amount of memory occupied at any moment in time for storing the items is bounded.

Syntax C

```
NBLListRoot* NBLListCreateLF_DB(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::List<T>* NBL::List<T>::CreateLF_DB(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the list. This measures approximately the number of items that can be stored into the list: $nrOfBlocks = n + N^2 * 14$, *where* n=nrOfItems, N=nrOfThreads

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a linked list object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.3.4 NBLListCreateLF_DU

Creates a new instance of a doubly linked list object using a lock-free implementation.

The amount of memory occupied at any moment in time for storing the items is unbounded.

Syntax C

```
NBLListRoot* NBLListCreateLF_DU(
  int nrOfBlocks
);
```

Syntax C++

```
template <typename T> class
static NBL::List<T>* NBL::List<T>::CreateLF_DU(
  int nrOfBlocks = 1000
);
```

Parameters

nrOfBlocks
[in] The maximum number of memory blocks that can be used for the implementation of the list. This measures approximately the number of items that can be stored into the list: $nrOfBlocks \geq n + N * 8$, *where* n=nrOfItems, N=nrOfThreads

On-line Parameters

PARAM_BACK_OFF_TYPE {BOT_LINEAR, BOT_EXPONENTIAL}
PARAM_BACK_OFF_INIT
PARAM_BACK_OFF_MAX

For detailed parameter descriptions see Section 2.4.3.

Return Values

If successful, the function returns a pointer to a new instance of a linked list object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.3.5 NBLListCreateLB_D

Creates a new instance of a doubly linked list object using a lock-based implementation.

Syntax C

```
NBLListRoot* NBLListCreateLB_D(
);
```

Syntax C++

```
template <typename T> class
static NBL::List<T>* NBL::List<T>::CreateLB_D(
);
```

Parameters

Return Values

If successful, the function returns a pointer to a new instance of a linked list object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4 Operations

10.4.1 NBLListInsertBefore

Inserts a new item directly before the current cursor position. If succesful, then positions the cursor to the newly inserted item.

Syntax C

```
int NBLListInsertBefore(
  NBLList* handle,
  void* item
);
```

Syntax C++

```
template <typename T> class
bool NBL::List<T>::InsertBefore(
  T* item
);
```

Parameters

handle
[in] A pointer to a local handle of a linked list object instance.

item
[in] Pointer to the item to insert in the list. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the list was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all inserted items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.2 NBLListInsertAfter

Inserts a new item directly after the current cursor position. If successful, then positions the cursor to the newly inserted item.

Syntax C

```
int NBLListInsertAfter(
  NBLList* handle,
  void* item
);
```

Syntax C++

```
template <typename T> class
bool NBL::List<T>::InsertAfter(
  T* item
);
```

Parameters

handle
: [in] A pointer to a local handle of a linked list object instance.

item
: [in] Pointer to the item to insert in the list. This should point to a memory resident structure which stays valid after the function call.

Return Values

If successful, the function returns true. Otherwise the list was full (i.e. out of memory) and the function returns false.

Remarks

Only the pointers to the new items are copied, which means that the user has to provide some kind of memory management system to be able to keep all inserted items valid.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.3 NBLListDelete

Removes the item at the current cursor position from the linked list.

Syntax C

```
void* NBLListDelete(
  NBLList* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::List<T>::Delete(
);
```

Parameters

handle
: [in] A pointer to a local handle of a linked list object instance.

Return Values

If successful, the function returns a pointer to the item that was removed from the list. Otherwise the item was already deleted or the cursor position invalid, and the function returns NULL.

Remarks

The actual item is not deleted itself by this function, and thus freeing and memory management has to be handled by the calling function.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.4 NBLListRead

Returns the item at the current cursor position in the linked list.

Syntax C

```
void* NBLListRead(
  NBLList* handle
);
```

Syntax C++

```
template <typename T> class
T* NBL::List<T>::Read(
);
```

Parameters

handle
: [in] A pointer to a local handle of a linked list object instance.

Return Values

If successful, the function returns a pointer to the item at the current cursor position. Otherwise the item has been deleted or the cursor position was invalid, and the function returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.5 NBLListFirst

Sets the current cursor position to point directly before the first item in the linked list.

Syntax C

```
void NBLListFirst(
  NBLList* handle
);
```

Syntax C++

```
template <typename T> class
void NBL::List<T>::First(
);
```

Parameters

handle
: [in] A pointer to a local handle of a linked list object instance.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.6 NBLListLast

Sets the current cursor position to point directly after the first item in the linked list.

Syntax C

```
void NBLListLast(
  NBLList* handle
);
```

Syntax C++

```
template <typename T> class
void NBL::List<T>::Last(
);
```

Parameters

handle
: [in] A pointer to a local handle of a linked list object instance.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.7 NBLListNext

Traverses the current cursor position one step forwards.

Syntax C

```
int NBLListNext(
  NBLList* handle
);
```

Syntax C++

```
template <typename T> class
bool NBL::List<T>::Next(
);
```

Parameters

handle
[in] A pointer to a local handle of a linked list object instance.

Return Values

If successful, the function returns true. Otherwise the cursor position has reached or was already positioned directly after the last item in the linked list.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

10.4.8 NBLListPrevious

Traverses the current cursor position one step forwards.

Syntax C

```
int NBLListPrevious(
  NBLList* handle
);
```

Syntax C++

```
template <typename T> class
bool NBL::List<T>::Previous(
);
```

Parameters

handle
[in] A pointer to a local handle of a linked list object instance.

Return Values

If successful, the function returns true. Otherwise the cursor position has reached or was already positioned directly before the first item in the linked list.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 11

Shared Snapshot

The Snapshot abstract data type is a collection of items where each item can be updated with new contents individually but read all together. The items are called components and are indexed uniquely. Basic operations are *Scan* (read the contents of all items) and *Update* (update an individual item with new contents).

The implementations available of a shared snapshot are:

- A Wait-Free implementation offering single scanner and single updater support.
- A Wait-Free implementation offering single scanner and multiple updaters support.
- A Wait-Free implementation for real-time systems offering single scanner and multiple updaters support.
- A Lock-Based implementation.

11.1 Overview

Here is a list of supported functions and macros together with the corresponding description.

11.1.1 C

NBLSnapshotCreateWF_SS Creates a new instance of a snapshot object using a wait-free implementation. See Section 11.3.1.

NBLSnapshotCreateWF_SM Creates a new instance of a snapshot object using a wait-free implementation. See Section 11.3.2.

NBLSnapshotCreateWFR_SM Creates a new instance of a snapshot object using a wait-free implementation for real-time systems. See Section 11.3.3.

NBLSnapshotCreateLB Creates a new instance of a snapshot object using a lock-based implementation. See Section 11.3.4.

NBLSnapshotScan Scans the components of the snapshot object. See Section 11.4.1.

NBLSnapshotUpdate Updates a single component of the snapshot object. See Section 11.4.2.

11.1.2 C++

```
namespace NBL {
  template <typename T> class Snapshot {
    // Constructors
    static Snapshot<T>* CreateWF_SS(int components);
    static Snapshot<T>* CreateWF_SM(int components, int writers
        );
    static Snapshot<T>* CreateWFR_SM(int components, int *
        cycles);
    static Snapshot<T>* CreateLB(int components);
    // Operations
    void Scan(T *values[]);
    void Update(int component, T *value);
  };
};
```

NBL::Snapshot::CreateWF_SS Creates a new instance of a snapshot object using a wait-free implementation. See Section 11.3.1.

NBL::Snapshot::CreateWF_SM Creates a new instance of a snapshot object using a wait-free implementation. See Section 11.3.2.

NBL::Snapshot::CreateWFR_SM Creates a new instance of a snapshot object using a wait-free implementation for real-time systems. See Section 11.3.3.

NBL::Snapshot::CreateLB Creates a new instance of a snapshot object using a lock-based implementation. See Section 11.3.4.

NBL::Snapshot::Scan Scans the components of the snapshot object. See Section 11.4.1.

NBL::Snapshot::Update Updates a single component of the snapshot object. See Section 11.4.2.

11.2 Examples

11.2.1 C

```
NBLSnapshotRoot *snapshot;
int values[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a wait-free snapshot object with 4 components */
  snapshot = NBLSnapshotCreateWF_SS(4);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the snapshot used */
  NBLSnapshotFree(snapshot);
  Snapshot=NULL;
}

void Thread1()
{
  NBLSnapshot *handle;
  /* Get the local handle to the shared snapshot */
  handle=NBLSnapshotGetHandle(snapshot);

  /* Scan the snapshot and get an atomic view of all components
       */
  NBLSnapshotScan(handle,values);

  /* Free the local handle to the shared snapshot */
  NBLSnapshotFreeHandle(handle);

}

void Thread2()
{
  NBLSnapshot *handle;
  /* Get the local handle to the shared Snapshot */
  handle=NBLSnapshotGetHandle(snapshot);

  /* Update component nr 0 with the value 5 */
  NBLSnapshotUpdate(handle,0,5);

  /* Free the local handle to the shared Snapshot */
  NBLSnapshotFreeHandle(handle);
}

void Thread3()
```

```
...
```

11.2.2 C++

```
NBL::Snapshot<int> *snapshot;
int values[4]={1,2,3,4};

void main(int argc, char **argv)
{
  /* Create a wait-free snapshot object with 4 components */
  snapshot = NBL::Snapshot<int>::CreateWF_SS(4);

  /* Creating and running the threads ... */
  ...

  /* Freeing the memory that the snapshot used */
  delete snapshot;
  snapshot=NULL;
}

void Thread1()
{
  /* Scan the snapshot and get an atomic view of all components
       */
  snapshot->Scan(values);

}

void Thread2()
{
  /* Update component nr 0 with the value 5 */
  snapshot->Update(0,5);

}

void Thread3()
...
```

11.3 Creation

The following functions are used to create a new Snapshot shared data structure.

11.3.1 NBLSnapshotCreateWF_SS

Creates a new instance of a snapshot object using a wait-free implementation.

The implementation supports concurrency up to a single scanner and a single updater per each component.

Syntax C

```
NBLSnapshotRoot* NBLSnapshotCreateWF_SS(
  int components
);
```

Syntax C++

```
template <typename T> class
static NBL::Snapshot<T>* NBL::Snapshot<T>::CreateWF_SS(
  int components
);
```

Parameters

components
[in] The number of components that the snapshot should consist of.

Return Values

If successful, the function returns a pointer to a new instance of a snapshot object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

11.3.2 NBLSnapshotCreateWF_SM

Creates a new instance of a snapshot object using a wait-free implementation.

The implementation supports concurrency up to a single scanner and multiple updaters per each component.

Syntax C

```
NBLSnapshotRoot* NBLSnapshotCreateWF_SM(
  int components,
  int writers
);
```

Syntax C++

```
template <typename T> class
static NBL::Snapshot<T>* NBL::Snapshot<T>::CreateWF_SM(
  int components,
  int writers
);
```

Parameters

components
: [in] The number of components that the snapshot should consist of.

writers
: [in] The maximum number of concurrent updaters to each component.

Return Values

If successful, the function returns a pointer to a new instance of a snapshot object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

11.3.3 NBLSnapshotCreateWFR_SM

Creates a new instance of a snapshot object using a wait-free implementation for real-time systems.

The implementation supports concurrency up to a single scanner and multiple updaters per each component.

Syntax C

```
NBLSnapshotRoot* NBLSnapshotCreateWFR_SM(
  int components,
  int *cycles,
);
```

Syntax C++

```
template <typename T> class
static NBL::Snapshot<T>* NBL::Snapshot<T>::CreateWFR_SM(
  int components,
  int *cycles,
);
```

Parameters

components
: [in] The number of components that the snapshot should consist of.

cycles
: [in] Array that gives the number of memory cells necessary for each component respectively.

Return Values

If successful, the function returns a pointer to a new instance of a snapshot object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

11.3.4 NBLSnapshotCreateLB

Creates a new instance of a snapshot object using a lock-based implementation.

Syntax C

```
NBLSnapshotRoot* NBLSnapshotCreateLB(
  int components
);
```

Syntax C++

```
template <typename T> class
static NBL::Snapshot<T>* NBL::Snapshot<T>::CreateLB(
  int components
);
```

Parameters

components
[in] The number of components that the snapshot should consist of.

Return Values

If successful, the function returns a pointer to a new instance of a snapshot object. Otherwise, it returns NULL.

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

11.4 Operations

11.4.1 NBLSnapshotScan

Scans the components of the snapshot data structure.

Syntax C

```
void NBLSnapshotScan(
  NBLSnapshot* handle,
  void** values
);
```

Syntax C++

```
template <typename T> class
void NBL::Snapshot<T>::Scan(
  T** values
);
```

Parameters

handle
: [in] A pointer to a local handle of a snapshot object instance.

values
: [out] Pointer to an srray that will contain pointers to the value objects for each component respectively.

Return Values

Remarks

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

11.4.2 NBLSnapshotUpdate

Updates a single component of the snapshot data structure.

Syntax C

```
void NBLSnapshotUpdate(
   NBLSnapshot* handle,
   int component,
   void *value
);
```

Syntax C++

```
template <typename T> class
void NBL::Snapshot<T>::Update(
   int component,
   T *value
);
```

Parameters

handle
: [in] A pointer to a local handle of a snapshot object instance.

handle
: [in] Index of the component which should be updated.

value
: [in] A pointer to the value object.

Return Values

Remarks

Only the pointer of the value is copied to the component, not the actual value object.

Requirements

Header: Declared in Noble.h; include Noble.h.
Library: Use libNOBLE.a (Unix) resp. Noble.lib (Windows).

See Also

Chapter 12

Configuration

12.1 Memory Allocation

These functions enable the user to change the memory allocation procedures used by NOBLE. The default memory allocation functions are the system calls *malloc* resp. *free*.

Here is a list of supported functions and macros together with the corresponding description.

NBLSetFunctionGlobalMemoryAlloc Changes the memory allocation function that NOBLE uses.

NBLSetFunctionGlobalMemoryFree Changes the memory free function that NOBLE uses.

NBLSetFunctionLocalMemoryAlloc Changes the memory allocation function that NOBLE uses.

NBLSetFunctionLocalMemoryFree Changes the memory free function that NOBLE uses.

Syntax

```
void NBLSetFunctionGlobalMemoryAlloc(
  void *(*alloc)(int size)
);

void NBLSetFunctionGlobalMemoryFree(
  void (*free)(void *mem)
);

void NBLSetFunctionLocalMemoryAlloc(
  void *(*alloc)(int size)
```

```
);

void NBLSetFunctionLocalMemoryFree(
  void (*free)(void *mem)
);
```

12.2 Mutual Exclusion

These functions enable the user to change the mutex handling used by NOBLE. The default mutex functions are based on spin-locks.

Here is a list of supported functions and macros together with the corresponding description.

NBLSetFunctionMutexCreate Changes the mutex creation function that NOBLE uses.

NBLSetFunctionMutexDelete Changes the mutex deletion function that NOBLE uses.

NBLSetFunctionMutexWait Changes the mutex wait function that NOBLE uses.

NBLSetFunctionMutexTryWait Changes the mutex trywait function that NOBLE uses.

NBLSetFunctionMutexSignal Changes the mutex signal function that NOBLE uses.

NBLSetFunctionSemaphoreCreate Changes the semaphore creation function that NOBLE uses.

NBLSetFunctionSemaphoreDelete Changes the semaphore deletion function that NOBLE uses.

NBLSetFunctionSemaphoreWait Changes the semaphore wait function that NOBLE uses.

NBLSetFunctionSemaphoreTryWait Changes the semaphore trywait function that NOBLE uses.

NBLSetFunctionSemaphoreSignal Changes the semaphore signal function that NOBLE uses.

Syntax

```
void NBLSetFunctionMutexCreate(
  void *(*create)()
);
```

```
void NBLSetFunctionMutexDelete(
  void (*del)(void *mutex)
);

void NBLSetFunctionMutexWait(
  void (*wait)(void *mutex)
);

void NBLSetFunctionMutexTryWait(
  int (*trywait)(void *mutex)
);

void NBLSetFunctionMutexSignal(
  void (*signal)(void *mutex)
);

void NBLSetFunctionSemaphoreCreate(
  void *(*create)(int init)
);

void NBLSetFunctionSemaphoreDelete(
  void (*del)(void *sem)
);

void NBLSetFunctionSemaphoreWait(
  void (*wait)(void *sem)
);

void NBLSetFunctionSemaphoreTryWait(
  int (*trywait)(void *sem)
);

void NBLSetFunctionSemaphoreSignal(
  void (*signal)(void *sem)
);
```

12.3 Examples

```
#include <semaphores.h>

void *CreateSem()
{
  sem_t *sem;
  sem=NBLMalloc(sizeof(sem_t));
  sem_init(sem,1,1);
  return sem;
}

void DeleteSem(void *sem)
```

```
{
  sem_destroy(sem);
  NBLFree(sem);
}

void WaitSem(void *sem)
{
  sem_wait(sem);
}

void SignalSem(void *sem)
{
  sem_post(sem);
}

void main(int argc, char **argv)
{
  /* Register the special mutex operations for NOBLE to use */
  NBLSetFunctionSemaphoreCreate(CreateSem);
  NBLSetFunctionSemaphoreDelete(DeleteSem);
  NBLSetFunctionSemaphoreWait(WaitSem);
  NBLSetFunctionSemaphoreSignal(SignalSem);
  /* Creating and running the threads ... */
  ...
}
```

PARALLEL©

SCALABLE SOLUTIONS

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