

uThreads

0.2.0

Generated by Doxygen 1.8.6

Sun May 8 2016 16:15:14

Contents

1	uThreads: Concurrent User-level Threads in C++	1
2	Hierarchical Index	5
2.1	Class Hierarchy	5
3	Class Index	7
3.1	Class List	7
4	File Index	9
4.1	File List	9
5	Class Documentation	11
5.1	BlockingQueue Class Reference	11
5.1.1	Detailed Description	12
5.1.2	Member Function Documentation	12
5.1.2.1	signal	12
5.1.2.2	signal	12
5.1.2.3	signal	12
5.1.2.4	signalAll	12
5.1.2.5	suspend	13
5.1.2.6	suspend	13
5.2	Cluster Class Reference	13
5.2.1	Detailed Description	14
5.2.2	Constructor & Destructor Documentation	14
5.2.2.1	Cluster	14
5.2.2.2	~Cluster	14
5.2.2.3	Cluster	14
5.2.3	Member Function Documentation	14
5.2.3.1	getDefaultCluster	14
5.2.3.2	getID	15
5.2.3.3	getNumberOfThreads	15
5.2.3.4	operator=	15
5.3	ConditionVariable Class Reference	15

5.3.1	Detailed Description	15
5.3.2	Member Function Documentation	16
5.3.2.1	empty	16
5.3.2.2	signal	16
5.3.2.3	signalAll	16
5.3.2.4	wait	16
5.4	Connection Class Reference	16
5.4.1	Detailed Description	17
5.4.2	Constructor & Destructor Documentation	18
5.4.2.1	Connection	18
5.4.2.2	Connection	18
5.4.2.3	Connection	18
5.4.2.4	Connection	18
5.4.3	Member Function Documentation	18
5.4.3.1	accept	18
5.4.3.2	accept	18
5.4.3.3	accept	19
5.4.3.4	bind	19
5.4.3.5	close	19
5.4.3.6	connect	19
5.4.3.7	getFd	19
5.4.3.8	listen	19
5.4.3.9	read	20
5.4.3.10	recv	20
5.4.3.11	recvfrom	20
5.4.3.12	recvmsg	20
5.4.3.13	recvmsg	21
5.4.3.14	send	21
5.4.3.15	sendmsg	21
5.4.3.16	sendmsg	22
5.4.3.17	sendto	22
5.4.3.18	socket	22
5.4.3.19	write	22
5.5	kThread Class Reference	23
5.5.1	Detailed Description	24
5.5.2	Constructor & Destructor Documentation	24
5.5.2.1	kThread	24
5.5.3	Member Function Documentation	24
5.5.3.1	currentkThread	24
5.5.3.2	getID	25

5.5.3.3	getThreadNativeHandle	25
5.5.3.4	getTotalNumberOfThreads	25
5.5.3.5	operator=	25
5.6	Mutex Class Reference	25
5.6.1	Detailed Description	26
5.6.2	Member Function Documentation	27
5.6.2.1	acquire	27
5.7	OwnerLock Class Reference	27
5.7.1	Detailed Description	28
5.7.2	Member Function Documentation	28
5.7.2.1	acquire	28
5.7.2.2	release	29
5.8	Semaphore Class Reference	29
5.8.1	Detailed Description	29
5.8.2	Constructor & Destructor Documentation	29
5.8.2.1	Semaphore	29
5.8.3	Member Function Documentation	29
5.8.3.1	P	29
5.9	uThread Class Reference	30
5.9.1	Detailed Description	32
5.9.2	Constructor & Destructor Documentation	32
5.9.2.1	uThread	32
5.9.3	Member Function Documentation	32
5.9.3.1	create	32
5.9.3.2	create	33
5.9.3.3	currentUThread	33
5.9.3.4	getCurrentCluster	33
5.9.3.5	getID	33
5.9.3.6	getTotalNumberOfUTs	33
5.9.3.7	join	34
5.9.3.8	migrate	34
5.9.3.9	operator=	34
5.9.3.10	start	34
5.9.3.11	terminate	34
5.9.3.12	yield	34
5.10	uThreadCache Class Reference	34
5.10.1	Detailed Description	35
5.10.2	Member Function Documentation	35
5.10.2.1	pop	35
5.10.2.2	push	35

6	File Documentation	37
6.1	src/cwrapper.h File Reference	37
6.1.1	Detailed Description	41
6.1.2	Function Documentation	41
6.1.2.1	cluster_create	41
6.1.2.2	cluster_destroy	41
6.1.2.3	cluster_get_current	41
6.1.2.4	cluster_get_default	41
6.1.2.5	cluster_get_id	41
6.1.2.6	cluster_get_number_of_kThreads	41
6.1.2.7	condition_variable_create	42
6.1.2.8	condition_variable_destroy	42
6.1.2.9	condition_variable_empty	42
6.1.2.10	condition_variable_signal	42
6.1.2.11	condition_variable_signal_all	42
6.1.2.12	condition_variable_wait	42
6.1.2.13	connection_accept	42
6.1.2.14	connection_accept_connexion	42
6.1.2.15	connection_accept_on_cluster	43
6.1.2.16	connection_bind	43
6.1.2.17	connection_block_on_read	43
6.1.2.18	connection_block_on_write	43
6.1.2.19	connection_close	43
6.1.2.20	connection_connect	43
6.1.2.21	connection_create	43
6.1.2.22	connection_create_on_cluster	44
6.1.2.23	connection_create_socket	45
6.1.2.24	connection_create_with_fd	45
6.1.2.25	connection_destory	45
6.1.2.26	connection_get_fd	45
6.1.2.27	connection_listen	45
6.1.2.28	connection_read	45
6.1.2.29	connection_recv	46
6.1.2.30	connection_recvfrom	46
6.1.2.31	connection_recvmsg	46
6.1.2.32	connection_recvmsg	46
6.1.2.33	connection_send	47
6.1.2.34	connection_sendmsg	47
6.1.2.35	connection_sendmsg	47
6.1.2.36	connection_sendto	48

6.1.2.37	connection_socket	48
6.1.2.38	connection_write	48
6.1.2.39	kThread_create	48
6.1.2.40	kThread_destroy	48
6.1.2.41	kThread_get_current	48
6.1.2.42	kThread_get_current_pthread_id	49
6.1.2.43	kThread_get_pthread_id	49
6.1.2.44	kThread_get_total_number_of_kThreads	49
6.1.2.45	mutex_acquire	49
6.1.2.46	mutex_create	49
6.1.2.47	mutex_destroy	49
6.1.2.48	mutex_release	49
6.1.2.49	ownerlock_acquire	49
6.1.2.50	ownerlock_create	49
6.1.2.51	ownerlock_destroy	49
6.1.2.52	ownerlock_release	49
6.1.2.53	semaphore_create	49
6.1.2.54	semaphore_destroy	50
6.1.2.55	semaphore_p	50
6.1.2.56	semaphore_v	50
6.1.2.57	uThread_create	50
6.1.2.58	uThread_detach	50
6.1.2.59	uThread_get_current	50
6.1.2.60	uThread_get_id	50
6.1.2.61	uThread_get_total_number_of_uThreads	51
6.1.2.62	uThread_join	51
6.1.2.63	uThread_migrate	51
6.1.2.64	uThread_start	51
6.1.2.65	uThread_terminate	51
6.1.2.66	uThread_yield	51
Index		52

Chapter 1

uThreads: Concurrent User-level Threads in C++

uThreads is a concurrent library based on cooperative scheduling of user-level threads implemented in C++. User-level threads are lightweight threads that execute on top of kernel threads to provide concurrency as well as parallelism. Kernel threads are necessary to utilize processors, but they come with the following drawbacks:

- Each suspend/resume operation involves a kernel context switch
- Thread preemption causes additional overhead
- Thread priorities and advanced scheduling causes additional overhead

Cooperative user-level threads, on the other hand, provide light weight context switches and omit the additional overhead of preemption and kernel scheduling. Most Operating Systems only support a 1:1 thread mapping (1 user-level thread to 1 kernel-level thread), where multiple kernel threads execute at the same time to utilize multiple cores and provide parallelism. e.g., Linux supports only 1:1 thread mapping. There is also N:1 thread mapping, where multiple user-level threads can be mapped to a single kernel-level thread. The kernel thread is not aware of the user-level threads existence. With N:1 mapping if the application blocks at the kernel level, it means blocking all user-level threads and application stops executing. This problem can be solved by using multiple kernel-level threads and map multiple user-level threads to each of them. Thus, creating the third scenario with M:N or hybrid mapping. e.g., **uC++** supports M:N mapping.

uThreads supports M:N mapping of *uThreads* (user-level threads) over *kThreads* (kernel-level threads) with cooperative scheduling. *kThreads* can be grouped together by *Clusters*, and uThreads can migrate among *Clusters*. Figure 1 shows the structure of an application implemented using uThreads. Each part is explained further in the following.

Clusters are used to group *kThreads* together. Each **Cluster** can contain one or more *kThreads*, but each **kThread** only belongs to a single **Cluster**. Each **Cluster** includes a single *ReadyQueue* which is used to schedule uThreads over *kThreads* in that **Cluster**. Application programmer decides how many *kThreads* belong to a *ReadyQueue* by assigning them to the related **Cluster**.

kThreads are kernel-level threads (`std::thread`), that are the main vehicle to utilize cores and execute the program. Each **kThread** can only pull uThreads from the *ReadyQueue* of the **Cluster** it belongs to, but it can push uThreads to the *ReadyQueue* of any **Cluster** in the application. The former can happen when uThreads *yield* or *block* at user level, and the latter happens when uThreads *migrate* to another **Cluster**. Migration let the code execute on a different set of *kThreads* based on the requirements of the code.

uThreads are the main building blocks of the library. They are either sitting in the *ReadyQueue* waiting to be picked by a **kThread**, running by a **kThread**, or blocked and waiting for an event to occur. uThreads are being scheduled cooperatively over *Clusters*, they can either yield, migrate or block on an event to let other uThreads utilized the same **kThread** they are being executed over.

Each application has at least one **Cluster**, one **kThread** and one **uThread**. Each C++ application has at least one thread of execution (kernel thread) which runs the *main()* function. A C++ application that is linked with uThreads library, upon execution, creates a *defaultCluster*, a wrapper around the main execution thread and call it *defaultkThread*, and also a **uThread** called *mainUT* to take over the *defaultkThread* stack and run the *main* function.

In addition, each [Cluster](#) by default has a *Poller kThread* which is responsible for polling the network devices, and multiplexing network events over the [Cluster](#). uThreads provide a user-level blocking network events, where network calls are non-blocking at the kernel-level but uThreads block on network events if the device is not ready for read/write. The poller thread is thus responsible for unblock the uThreads upon receiving the related network event. The poller thread is using *edge triggered epoll* in Linux, and the model is similar to [Golang](#).

By default there is a [uThread](#) cache to cache uThreads that finished executing and avoid the extra overhead of memory allocation. Currently, this cache only supports uThreads with same stack size and does not support the scenario where stack sizes are different. This feature will be added in the near future.

Migration and Joinable uThreads

uThreads can be joinable, where upon creating the creator has to wait for them to finish execution and join with them. So there are two ways to execute a piece of code on another [Cluster](#):

- **Migration:** [uThread](#) can migrate to another [Cluster](#) to execute a piece of code and it can either migrate back to the previous [Cluster](#) or continue the execution on the same [Cluster](#) or migrate to a different [Cluster](#). The following code demonstrates a simple scenario to migrate to a different cluster and back, assuming [uThread](#) is executing on the *defaultCluster*.

```
Cluster *cluster1;

void func(){
    // some code
    migrate(*cluster1);
    // code to run on cluster1
    migrate(Cluster::getDefaultCluster());
    // some more code
}

int main(){

    cluster1 = new Cluster();
    kThread kt(*cluster1);
    uThread *ut = uThread::create();
    ut->start(Cluster::getDefaultCluster(), func);
    .
    .
    .
}
```

- **Joinable thread:** Create a joinable thread on the remote [Cluster](#) and wait for it to finish execution. While waiting, the [uThread](#) is blocked at user-level and will be unblocked by the newly created [uThread](#).

```
Cluster *cluster1;

void run(){
    //code to run on cluster1
}

void func(){
    // some code
    uThread *ut2 = uThread::create(true); //create a joinable thread
    ut2->start(cluster1, run);
    ut2->join(); //wait for ut2 to finish execution and join
    // some more code
}

int main(){

    cluster1 = new Cluster();
    kThread kt(*cluster1);
    uThread *ut = uThread::create();
    ut->start(Cluster::getDefaultCluster(), func);
    .
    .
    .
}
```

User-level Blocking Synchronization Primitives

uThreads also provides user-level blocking synchronization and mutex primitives. It has basic [Mutex](#), Condition Variable and [Semaphore](#). You can find examples of their usage under *test* directory in the [github repo](#).

Examples

You can find various examples under the *test* directory in the [github repo](#). There is an [EchoClient](#) and [EchoServer](#) implemented using uThreads.

There is also a simple [webserver](#) to test uThreads functionality.

For performance comparisons, memached code has been updated to use uThreads instead of event loops (except the thread that accepts connections), where tasks are assigned to uThreads instead of using the underlying event library. The code can be found [here](#).

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

BlockingQueue	11
Cluster	13
ConditionVariable	15
Connection	16
Link	
kThread	23
uThread	30
Mutex	25
OwnerLock	27
Semaphore	29
uThreadCache	34

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

BlockingQueue	A queue used to keep track of blocked uThreads	11
Cluster	Scheduler and Cluster of kThreads	13
ConditionVariable	A user level condition variable	15
Connection	Represents a network connection	16
kThread	Object to represent kernel threads	23
Mutex	A user-level Mutex	25
OwnerLock	Owner Mutex where owner can recursively acquire the Mutex	27
Semaphore	A user-level Semaphore	29
uThread	User-level threads (fiber)	30
uThreadCache	Data structure to cache uThreads	34

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

src/ cwrapper.h	
C Wrapper for uThreads	37
src/io/ IOHandler.h	??
src/io/ Network.h	??
src/runtime/ BlockingSync.h	??
src/runtime/ Cluster.h	??
src/runtime/ kThread.h	??
src/runtime/ Stack.h	??
src/runtime/ uThread.h	??
src/runtime/ uThreadCache.h	??

Chapter 5

Class Documentation

5.1 BlockingQueue Class Reference

A queue used to keep track of blocked uThreads.

```
#include <BlockingSync.h>
```

Public Member Functions

- **BlockingQueue** (const **BlockingQueue** &)=delete
BlockingQueue cannot be copied or assigned.
- const **BlockingQueue** & **operator=** (const **BlockingQueue** &)=delete
BlockingQueue cannot be copied or assigned.
- bool **empty** () const
Whether the queue is empty or not.
- bool **suspend** (std::mutex &lock)
Suspends the uThread and add it to the queue.
- bool **suspend** (**Mutex** &)
Suspends the uThread and add it to the queue.
- bool **signal** (std::mutex &lock, **uThread** *&owner)
Unblock one blocked uThread, used for OwnerLock.
- bool **signal** (std::mutex &lock)
unblock one blocked, used for Mutex
- bool **signal** (**Mutex** &)
unblock one blocked uThread, used for ConditionVariable
- void **signalAll** (**Mutex** &)
unblock all blocked uThreads, used for Condition Variable

Static Public Member Functions

- template<typename T >
static void **postSwitchFunc** (void *ut, void *args)

Friends

- class **uThread**
- class **Mutex**

- class **OwnerLock**
- class **ConditionVariable**
- class **Semaphore**

5.1.1 Detailed Description

A queue used to keep track of blocked uThreads.

This queue is a FIFO queue used to hold blocked uThreads on [Mutex](#), [Semaphore](#), or Condition Variable.

5.1.2 Member Function Documentation

5.1.2.1 `bool BlockingQueue::signal (std::mutex & lock, uThread *& owner)`

Unblock one blocked [uThread](#), used for [OwnerLock](#).

Parameters

<i>lock</i>	mutex to be released after signal is done
<i>owner</i>	passed to support atomic setting of Mutex::owner

Returns

true if a [uThread](#) was unblocked, and false otherwise

5.1.2.2 `bool BlockingQueue::signal (std::mutex & lock) [inline]`

unblock one blocked, used for [Mutex](#)

Parameters

<i>lock</i>	mutex to be released after signal is done
-------------	---

Returns

true if a [uThread](#) was unblocked, and false otherwise

5.1.2.3 `bool BlockingQueue::signal (Mutex & mutex)`

unblock one blocked [uThread](#), used for [ConditionVariable](#)

Parameters

Mutex	that is released after signal is done
-----------------------	---------------------------------------

Returns

true if a [uThread](#) was unblocked, and false otherwise

5.1.2.4 `void BlockingQueue::signalAll (Mutex & mutex)`

unblock all blocked uThreads, used for Condition Variable

Parameters

Mutex	to be released after signalAll is done
-----------------------	--

5.1.2.5 bool BlockingQueue::suspend (std::mutex & lock)

Suspends the [uThread](#) and add it to the queue.

Parameters

<i>lock</i>	a mutex to be released after blocking
-------------	---------------------------------------

Returns

whether the suspension was successful or not

Suspends the [uThread](#) and adds it to the [BlockingQueue](#).

5.1.2.6 bool BlockingQueue::suspend (Mutex & mutex)

Suspends the [uThread](#) and add it to the queue.

Parameters

<i>lock</i>	a mutex to be released after blocking
-------------	---------------------------------------

Returns

whether the suspension was successful or not

Suspends the [uThread](#) and adds it to the [BlockingQueue](#).

The documentation for this class was generated from the following files:

- src/runtime/BlockingSync.h
- src/runtime/BlockingSync.cpp

5.2 Cluster Class Reference

Scheduler and [Cluster](#) of kThreads.

```
#include <Cluster.h>
```

Public Member Functions

- [Cluster](#) ()
- virtual [~Cluster](#) ()
- [Cluster](#) (const [Cluster](#) &)=delete
- const [Cluster](#) & [operator=](#) (const [Cluster](#) &)=delete
- uint64_t [getID](#) () const
Get the ID of [Cluster](#).
- size_t [getNumberOfkThreads](#) () const
Total number of kThreads belonging to this cluster.
- IOHandler * [getIOHandler](#) ()

Static Public Member Functions

- static [Cluster](#) & [getDefaultCluster](#) ()

Friends

- class **kThread**
- class **uThread**
- class **Connection**
- class **IOHandler**

5.2.1 Detailed Description

Scheduler and [Cluster](#) of kThreads.

[Cluster](#) is an entity that contains multiple kernel threads ([kThread](#)). Each cluster is responsible for maintaining a ready queue and performing basic scheduling tasks. Programs can have as many Clusters as is necessary. The [Cluster](#)'s ReadyQueue is a multiple-producer multiple-consumer queue where consumers are only kThreads belonging to that [Cluster](#), and producers can be any running [kThread](#). kThreads constantly push and pull uThreads to/from the ReadyQueue. [Cluster](#) is an interface between kThreads and the ReadyQueue, and also provides the means to group kThreads together.

Each [Cluster](#) has its own IOHandler. IOHandler is responsible for providing asynchronous nonblocking access to IO devices. For now each instance of an IOHandler has its own dedicated poller thread, which means each cluster has a dedicated IO poller thread when it is created. This might change in the future. Each [uThread](#) that requires access to IO uses the IOHandler to avoid blocking the [kThread](#), if the device is ready for read or write, the [uThread](#) continues otherwise it blocks until it is ready, and the [kThread](#) execute another [uThread](#) from the ReadyQueue.

When the program starts a defaultCluster is created for the kernel thread that runs the *main* function. defaultCluster can be used like any other clusters.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 [Cluster::Cluster](#) ()

Create a new [Cluster](#)

5.2.2.2 [Cluster::~~Cluster](#) () [virtual]

Destroy the cluster

5.2.2.3 [Cluster::Cluster](#) (const [Cluster](#) &) [delete]

[Cluster](#) cannot be copied or assigned.

5.2.3 Member Function Documentation

5.2.3.1 static [Cluster](#)& [Cluster::getDefaultCluster](#) () [inline],[static]

Returns

defaultCluster

5.2.3.2 uint64_t Cluster::getID () const [inline]

Get the ID of [Cluster](#).

Returns

The ID of the cluster

5.2.3.3 size_t Cluster::getNumberOfThreads () const [inline]

Total number of kThreads belonging to this cluster.

Returns

Total number of kThreads belonging to this cluster

5.2.3.4 const Cluster& Cluster::operator= (const Cluster &) [delete]

[Cluster](#) cannot be copied or assigned.

The documentation for this class was generated from the following files:

- src/runtime/Cluster.h
- src/runtime/Cluster.cpp
- src/runtime/uThread.cpp

5.3 ConditionVariable Class Reference

A user level condition variable.

```
#include <BlockingSync.h>
```

Public Member Functions

- void [wait](#) ([Mutex](#) &mutex)
Block [uThread](#) on the condition variable using the provided mutex.
- void [signal](#) ([Mutex](#) &mutex)
Unblock one [uThread](#) waiting on the condition variable.
- void [signalAll](#) ([Mutex](#) &mutex)
unblock all [uThreads](#) waiting on the condition variable
- bool [empty](#) ()
Whether the waiting list is empty or not.

5.3.1 Detailed Description

A user level condition variable.

User-level Condition Variable blocks only in user-level by suspending the uThreads instead of blocking the kernel threads.

5.3.2 Member Function Documentation

5.3.2.1 `bool ConditionVariable::empty () [inline]`

Whether the waiting list is empty or not.

Returns

Whether the waiting list is empty or not

5.3.2.2 `void ConditionVariable::signal (Mutex & mutex) [inline]`

Unblock one [uThread](#) waiting on the condition variable.

Parameters

<i>mutex</i>	The mutex to be released after unblocking is done
--------------	---

5.3.2.3 `void ConditionVariable::signalAll (Mutex & mutex) [inline]`

unblock all uThreads waiting on the condition variable

Parameters

<i>mutex</i>	The mutex to be released after unblocking is done
--------------	---

5.3.2.4 `void ConditionVariable::wait (Mutex & mutex) [inline]`

Block [uThread](#) on the condition variable using the provided mutex.

Parameters

<i>mutex</i>	used to synchronize access to the condition
--------------	---

The documentation for this class was generated from the following file:

- `src/runtime/BlockingSync.h`

5.4 Connection Class Reference

Represents a network connection.

```
#include <Network.h>
```

Public Member Functions

- [Connection](#) ()
Create a [Connection](#) that does not have.
- [Connection](#) (int fd)
Create a connection object with the provided fd.
- [Connection](#) ([Cluster](#) &cluster, int fd)
Create a connection object with the provided fd, and add it to the poller thread of the provided cluster.
- [Connection](#) (int domain, int type, int protocol) throw (std::system_error)
Same as socket syscall adds | SOCK_NONBLOCK to type.

- `int accept (Connection *conn, struct sockaddr *addr, socklen_t *addrlen)`
nonblocking accept syscall and updating the passed [Connection](#) object
- `Connection * accept (struct sockaddr *addr, socklen_t *addrlen) throw (std::system_error)`
Accepts a connection and returns a connection object.
- `Connection * accept (Cluster &cluster, struct sockaddr *addr, socklen_t *addrlen) throw (std::system_error)`
Accepts a connection, adds it to the poller thread of the provided cluster, and returns a connection object.
- `int socket (int domain, int type, int protocol)`
Same as socket syscall, set the fd for current connection.
- `int listen (int backlog)`
Same as listen syscall on current fd.
- `int bind (const struct sockaddr *addr, socklen_t addrlen)`
Same as bind syscall.
- `int connect (const struct sockaddr *addr, socklen_t addrlen)`
Same as connect syscall.
- `ssize_t recv (void *buf, size_t len, int flags)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t recvfrom (void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t recvmsg (int sockfd, struct msghdr *msg, int flags)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `int recvmsg (int sockfd, struct mmsghdr *msgvec, unsigned int vlen, unsigned int flags, struct timespec *timeout)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t send (const void *buf, size_t len, int flags)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t sendto (int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t sendmsg (const struct msghdr *msg, int flags)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `int sendmsg (int sockfd, struct mmsghdr *msgvec, unsigned int vlen, unsigned int flags)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t read (void *buf, size_t count)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `ssize_t write (const void *buf, size_t count)`
Call the underlying system call on [Connection](#)'s file descriptor.
- `void blockOnRead ()`
Block [uThread](#), waiting for fd to become ready for read.
- `void blockOnWrite ()`
Block [uThread](#), waiting for fd to become ready for write.
- `int close ()`
closes the socket
- `int getFd () const`
return the [Connection](#)'s file descriptor

5.4.1 Detailed Description

Represents a network connection.

[Connection](#) class is a wrapper around socket and provides the ability to do nonblocking read/write on sockets, and nonblocking accept. It first tries to read/write/accept and if the fd is not ready uses the underlying polling structure to wait for the fd to be ready. Thus, the [uThread](#) that is calling these functions is blocked if the fd is not ready, and [kThread](#) never blocks.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 `Connection::Connection () [inline]`

Create a [Connection](#) that does not have.

This is useful for accept or socket functions that require a [Connection](#) object without fd being set

5.4.2.2 `Connection::Connection (int fd) [inline]`

Create a connection object with the provided fd.

Parameters

<i>fd</i>	If the connection is already established by other means, set the fd and add it to the polling structure
-----------	---

5.4.2.3 `Connection::Connection (Cluster & cluster, int fd) [inline]`

Create a connection object with the provided fd, and add it to the poller thread of the provided cluster.

Parameters

<i>fd</i>	If the connection is already established by other means, set the fd and add it to the polling structure
-----------	---

5.4.2.4 `Connection::Connection (int domain, int type, int protocol) throw std::system_error`

Same as socket syscall adds | SOCK_NONBLOCK to type.

Returns

same as socket syscall

Throws a std::system_error exception. Do not call from C code. The unerlying socket is always nonblocking. This is achieved by adding a (| SOCK_NONBLOCK) to type, thus requires linux kernels > 2.6.27

5.4.3 Member Function Documentation

5.4.3.1 `int Connection::accept (Connection * conn, struct sockaddr * addr, socklen_t * addrlen)`

nonblocking accept syscall and updating the passed [Connection](#) object

Parameters

<i>conn</i>	Pointer to a Connection object that is not initialized
-------------	--

Returns

same as accept system call

This format is used for compatibility with C

5.4.3.2 `Connection * Connection::accept (struct sockaddr * addr, socklen_t * addrlen) throw std::system_error`

Accepts a connection and returns a connection object.

Returns

Newly created connection

Throws a `std::system_error` exception on error. Never call from C.

5.4.3.3 `Connection * Connection::accept (Cluster & cluster, struct sockaddr * addr, socklen_t * addrlen) throw std::system_error`

Accepts a connection, adds it to the poller thread of the provided cluster, and returns a connection object.

Returns

Newly created connection

Throws a `std::system_error` exception on error. Never call from C.

5.4.3.4 `int Connection::bind (const struct sockaddr * addr, socklen_t addrlen)`

Same as `bind` syscall.

Returns

Same as `bind` syscall

5.4.3.5 `int Connection::close ()`

closes the socket

Returns

the same as `close` system call

5.4.3.6 `int Connection::connect (const struct sockaddr * addr, socklen_t addrlen)`

Same as `connect` syscall.

Returns

Same as `connect` syscall

5.4.3.7 `int Connection::getFd () const [inline]`

return the [Connection](#)'s file descriptor

Returns

file descriptor

5.4.3.8 `int Connection::listen (int backlog)`

Same as `listen` syscall on current fd.

Returns

Same as `listen` syscall

5.4.3.9 `ssize_t Connection::read (void * buf, size_t count)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.10 `ssize_t Connection::recv (void * buf, size_t len, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.11 `ssize_t Connection::recvfrom (void * buf, size_t len, int flags, struct sockaddr * src_addr, socklen_t * addrlen)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.12 `int Connection::recvmsg (int sockfd, struct mmsghdr * msgvec, unsigned int vlen, unsigned int flags, struct timespec * timeout)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related systemcall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.13 `ssize_t Connection::recvmsg (int sockfd, struct msghdr * msg, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related systemcall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.14 `ssize_t Connection::send (const void * buf, size_t len, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related systemcall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.15 `int Connection::sendmmsg (int sockfd, struct mmsghdr * msgvec, unsigned int vlen, unsigned int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related systemcall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.16 `ssize_t Connection::sendmsg (const struct msghdr * msg, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.17 `ssize_t Connection::sendto (int sockfd, const void * buf, size_t len, int flags, const struct sockaddr * dest_addr, socklen_t addrlen)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

5.4.3.18 `int Connection::socket (int domain, int type, int protocol)`

Same as socket syscall, set the fd for current connection.

Returns

same as socket syscall The underlying socket is always nonblocking. This is achieved by adding a `(| SOCK_ - NONBLOCK)` to type, thus requires linux kernels $> 2.6.27$

5.4.3.19 `ssize_t Connection::write (const void * buf, size_t count)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

The documentation for this class was generated from the following files:

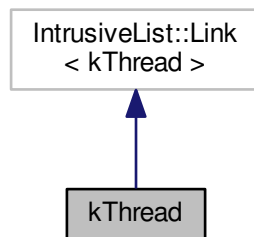
- `src/io/Network.h`
- `src/io/Network.cpp`

5.5 kThread Class Reference

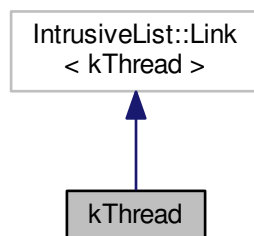
Object to represent kernel threads.

```
#include <kThread.h>
```

Inheritance diagram for kThread:



Collaboration diagram for kThread:



Public Member Functions

- `kThread (Cluster &)`
Create a `kThread` on the passed cluster.
- `kThread (const kThread &)=delete`
`kThread` cannot be copied or assigned.
- `const kThread & operator= (const kThread &)=delete`
`kThread` cannot be copied or assigned.
- `std::thread::native_handle_type getThreadNativeHandle ()`
return the native handle for the kernel thread

- `std::thread::id` `getID ()`
returns the kernel thread ID

Static Public Member Functions

- static `kThread` * `currentkThread ()`
Get the pointer to the current `kThread`.
- static uint `getTotalNumberOfkThreads ()`

Friends

- class `uThread`
- class `Cluster`
- class `ReadyQueue`
- class `IOHandler`

5.5.1 Detailed Description

Object to represent kernel threads.

`kThread` is an interface for underlying kernel threads. `kThreads` are pulling and pushing `uThreads` from ReadyQueue provided by the `Cluster` and context switch to them and execute them. Each `kThread` belongs to only and only one `Cluster` and it can only pull `uThreads` from the ReadyQueue of that `Cluster`. However, `kThreads` can push `uThreads` to the ReadyQueue of any cluster.

`defaultKT` is the first kernel thread that executes and is responsible for running the `main()` function. `defaultKT` is created when the program starts.

Each `kThread` has a `mainUT` which is a `uThread` used when the ReadyQueue is empty. `mainUT` is used to switch out the previous `uThread` and either pull `uThreads` from the ReadyQueue if it's not empty, or block on a condition variable waiting for `uThreads` to be pushed to the queue.

`kThreads` can be created by passing a `Cluster` to the constructor of the `kThread`.

5.5.2 Constructor & Destructor Documentation

5.5.2.1 `kThread::kThread (Cluster & cluster)`

Create a `kThread` on the passed cluster.

Parameters

<i>The</i>	<code>Cluster</code> this <code>kThread</code> belongs to.
------------	--

5.5.3 Member Function Documentation

5.5.3.1 `static kThread* kThread::currentkThread () [inline],[static]`

Get the pointer to the current `kThread`.

Returns

current `kThread`

This is necessary when a `uThread` wants to find which `kThread` it is being executed over.

5.5.3.2 `std::thread::id kThread::getID ()`

returns the kernel thread ID

Returns

the [kThread](#) ID

The returned type depends on the platform.

5.5.3.3 `std::thread::native_handle_type kThread::getThreadNativeHandle ()`

return the native handle for the kernel thread

Returns

native handle for the [kThread](#)

In linux this is `pthread_t` representation of the thread.

5.5.3.4 `static uint kThread::getTotalNumberOfkThreads () [inline],[static]`

Returns

total number of `kThreads` running under the program.

5.5.3.5 `const kThread& kThread::operator= (const kThread &) [delete]`

[kThread](#) cannot be copied or assigned.

The documentation for this class was generated from the following files:

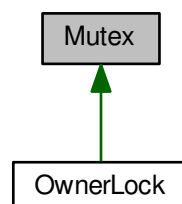
- `src/runtime/kThread.h`
- `src/runtime/kThread.cpp`
- `src/runtime/uThread.cpp`

5.6 Mutex Class Reference

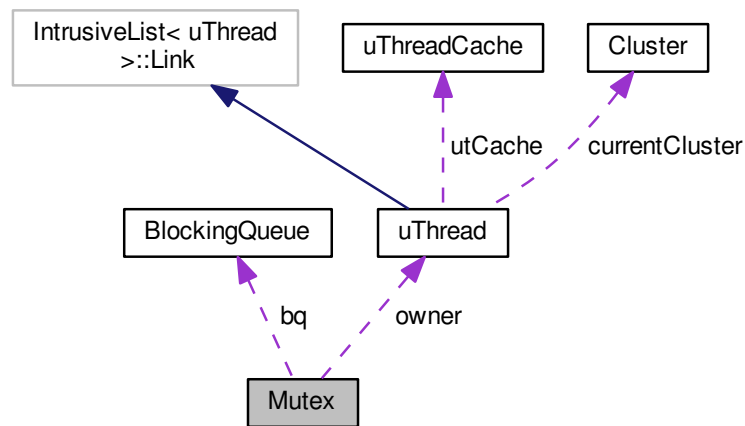
A user-level [Mutex](#).

```
#include <BlockingSync.h>
```

Inheritance diagram for `Mutex`:



Collaboration diagram for Mutex:



Public Member Functions

- `template<bool OL = false>`
`bool acquire ()`
acquire the mutex
- `void release ()`
release the [Mutex](#)
- `void unlock ()`

Protected Member Functions

- `template<bool OL>`
`bool internalAcquire (mword timeout)`
- `void internalRelease ()`

Protected Attributes

- `std::mutex lock`
- `BlockingQueue bq`
- `uThread * owner`

Friends

- class `ConditionVariable`
- class `Semaphore`
- class `BlockingQueue`

5.6.1 Detailed Description

A user-level [Mutex](#).

5.6.2 Member Function Documentation

5.6.2.1 `template<bool OL = false> bool Mutex::acquire () [inline]`

acquire the mutex

Returns

true if it was acquired, false otherwise

The return value is only for when timeouts are implemented

The documentation for this class was generated from the following file:

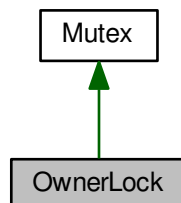
- `src/runtime/BlockingSync.h`

5.7 OwnerLock Class Reference

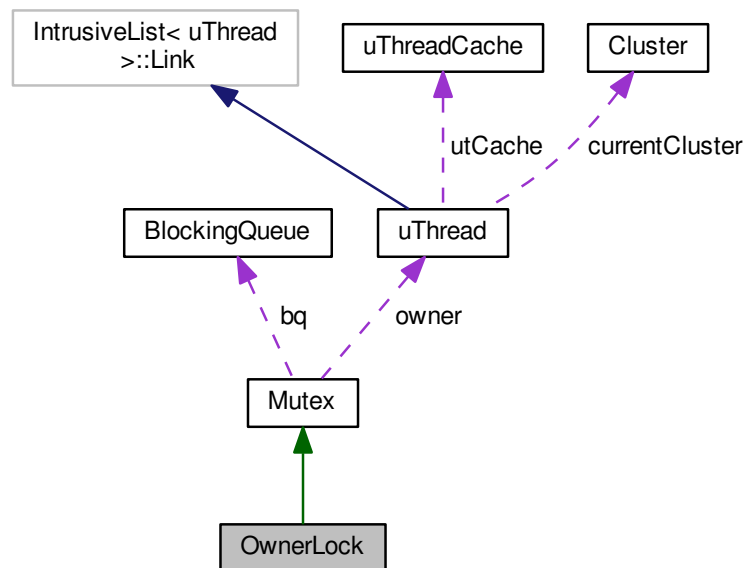
an Owner [Mutex](#) where owner can recursively acquire the [Mutex](#)

```
#include <BlockingSync.h>
```

Inheritance diagram for OwnerLock:



Collaboration diagram for OwnerLock:



Public Member Functions

- `OwnerLock ()`
Create a new *OwnerLock*.
- mword `acquire ()`
Acquire the *OwnerLock*.
- mword `release ()`
Release the *OwnerLock* once.

Additional Inherited Members

5.7.1 Detailed Description

an Owner `Mutex` where owner can recursively acquire the `Mutex`

5.7.2 Member Function Documentation

5.7.2.1 mword OwnerLock::acquire () [inline]

Acquire the *OwnerLock*.

Returns

The number of times current owner acquired the lock

5.7.2.2 mword OwnerLock::release() [inline]

Release the [OwnerLock](#) once.

Returns

The number of times current owner acquired the lock, if lock is released completely the result is 0

The documentation for this class was generated from the following file:

- src/runtime/BlockingSync.h

5.8 Semaphore Class Reference

A user-level [Semaphore](#).

```
#include <BlockingSync.h>
```

Public Member Functions

- [Semaphore](#) (mword c=0)
Create a new [Semaphore](#).
- bool [P](#) ()
Decrement the value of the [Semaphore](#).
- void [V](#) ()
increment the value of the [Semaphore](#)

5.8.1 Detailed Description

A user-level [Semaphore](#).

blocks in user level by blocking uThreads and does not cause kernel threads to block.

5.8.2 Constructor & Destructor Documentation

5.8.2.1 Semaphore::Semaphore (mword c = 0) [inline], [explicit]

Create a new [Semaphore](#).

Parameters

c	Initial value of the Semaphore
---	--

5.8.3 Member Function Documentation

5.8.3.1 bool Semaphore::P () [inline]

Decrement the value of the [Semaphore](#).

Returns

Whether it was successful or not

The documentation for this class was generated from the following file:

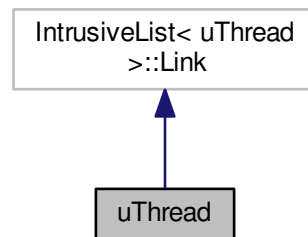
- src/runtime/BlockingSync.h

5.9 uThread Class Reference

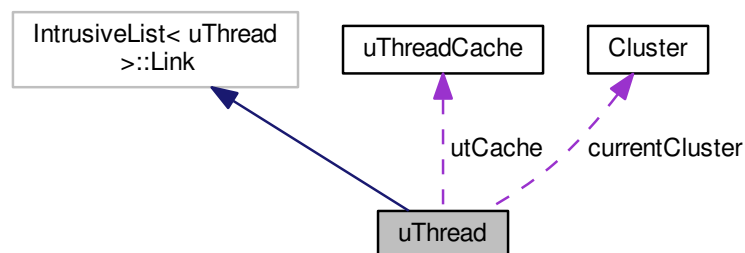
user-level threads (fiber)

```
#include <uThread.h>
```

Inheritance diagram for uThread:



Collaboration diagram for uThread:



Public Member Functions

- `uThread (const uThread &)=delete`
uThread cannot be copied or assigned
- `const uThread & operator= (const uThread &)=delete`
- `void start (const Cluster &cluster, ptr_t func, ptr_t arg1=nullptr, ptr_t arg2=nullptr, ptr_t arg3=nullptr)`
start the uThread by calling the function passed to it
- `void resume ()`
Resumes the uThread. If uThread is blocked or is waiting on IO it will be placed back on the ReadyQueue.
- `bool join ()`
Wait for uThread to finish execution and exit.
- `void detach ()`
Detach a joinable thread.
- `Cluster & getCurrentCluster () const`

return the current *Cluster uThread* is executed on

- uint64_t **getID** () const

get the ID of this *uThread*

Static Public Member Functions

- static *uThread* * **create** (size_t ss, bool joinable=false)

Create a *uThread* with a given stack size.

- static *uThread* * **create** (bool joinable=false)

Create a *uThread* with default stack size.

- static void **yield** ()

Causes *uThread* to yield.

- static void **terminate** ()

Terminates the *uThread*.

- static void **migrate** (*Cluster* *)

Move the *uThread* to the provided cluster.

- static uint64_t **getTotalNumberOfUTs** ()

- static *uThread* * **currentUThread** ()

Get a pointer to the current running *uThread*.

Protected Types

- enum **State** : std::uint8_t {
 INITIALIZED, **READY**, **RUNNING**, **YIELD**,
 MIGRATE, **WAITING**, **TERMINATED** }
- enum **JState** : std::uint8_t { **DETACHED**, **JOINABLE**, **JOINING** }

Protected Member Functions

- *uThread* (*vaddr* sb, size_t ss)
- virtual void **destory** (bool)
- void **reset** ()
- void **suspend** (*funcvoid2_t*, void *)

Static Protected Member Functions

- static *vaddr* **createStack** (size_t)
- static void **invoke** (*funcvoid3_t*, *ptr_t*, *ptr_t*, *ptr_t*) __noreturn

Protected Attributes

- enum *uThread::State* **state**
- enum *uThread::JState* **jState**
- uint64_t **uThreadID**
- *Cluster* * **currentCluster**
- size_t **stackSize**
- *vaddr* **stackPointer**
- *vaddr* **stackBottom**

Static Protected Attributes

- static [uThreadCache](#) **utCache**
- static std::atomic_ulong **totalNumberOfUTs**
- static std::atomic_ulong **uThreadMasterID**

Friends

- class **uThreadCache**
- class **kThread**
- class **Cluster**
- class **BlockingQueue**
- class **IOHandler**

5.9.1 Detailed Description

user-level threads (fiber)

uThreads are building blocks of this library. They are lightweight threads and do not have the same context switch overhead as kernel threads. Each [uThread](#) is an execution unit provided to run small tasks. uThreads are being managed cooperatively and there is no preemption involved. uThreads either yield, migrate, or blocked and giving way to other uThreads to get a chance to run.

Due to the cooperative nature of uThreads, it is recommended that uThreads do not block the underlying kernel thread for a long time. However, since there can be multiple kernel threads ([kThread](#)) in the program, if one or more uThreads block underlying kThreads for small amount of time the execution of the program does not stop and other kThreads keep executing.

Another pitfall can be when all uThreads are blocked and each waiting for an event to occurs which can cause deadlocks. It is programmer's responsibility to make sure this never happens. Although it never happens unless a [uThread](#) is blocked without a reason (not waiting on a lock or IO), otherwise there is always an event (another [uThread](#) or polling structure) that unblock the [uThread](#).

Each [uThread](#) has its own stack which is very smaller than kernel thread's stack. This stack is allocated when [uThread](#) is created.

5.9.2 Constructor & Destructor Documentation

5.9.2.1 `uThread::uThread (vaddr sb, size_t ss) [inline], [protected]`

The main and only constructor for [uThread](#). uThreads are not supposed to be created by using the constructor. The memory used to save the [uThread](#) object is allocated at the beginning of its own stack. Thus, by freeing the stack memory [uThread](#) object is being destroyed as well. Therefore, the implicit destructor is not necessary.

5.9.3 Member Function Documentation

5.9.3.1 `uThread * uThread::create (size_t ss, bool joinable = false) [static]`

Create a [uThread](#) with a given stack size.

Parameters

<code>ss</code>	stack size
-----------------	------------

<i>joinable</i>	Whether this thread is joinable or detached
-----------------	---

Returns

a pointer to a new [uThread](#)

This function relies on a [uThreadCache](#) structure and does not always allocate the stack.

5.9.3.2 `static uThread* uThread::create (bool joinable = false) [inline],[static]`

Create a [uThread](#) with default stack size.

Parameters

<i>joinable</i>	Whether this thread is joinable or detached
-----------------	---

Returns

a pointer to a new [uThread](#)

5.9.3.3 `uThread * uThread::currentUThread () [static]`

Get a pointer to the current running [uThread](#).

Returns

pointer to the current [uThread](#)

5.9.3.4 `Cluster& uThread::getCurrentCluster () const [inline]`

return the current [Cluster](#) [uThread](#) is executed on

Returns

the current [Cluster](#) [uThread](#) is executed on

5.9.3.5 `uint64_t uThread::getID () const [inline]`

get the ID of this [uThread](#)

Returns

ID of the [uThread](#)

5.9.3.6 `static uint64_t uThread::getTotalNumberOfUTs () [inline],[static]`

Returns

Total number of uThreads in the program

This number does not include mainUT or IOUTs

5.9.3.7 `bool uThread::join ()`

Wait for `uThread` to finish execution and exit.

Returns

Whether join was successful or failed

5.9.3.8 `void uThread::migrate (Cluster * cluster) [static]`

Move the `uThread` to the provided cluster.

Parameters

<i>cluster</i>	This function is used to migrate the <code>uThread</code> to another <code>Cluster</code> . Migration is useful specially if clusters form a pipeline of execution.
----------------	---

5.9.3.9 `const uThread& uThread::operator= (const uThread &) [delete]`

5.9.3.10 `void uThread::start (const Cluster & cluster, ptr_t func, ptr_t arg1 = nullptr, ptr_t arg2 = nullptr, ptr_t arg3 = nullptr)`

start the `uThread` by calling the function passed to it

Parameters

<i>cluster</i>	The cluster that function belongs to.
<i>func</i>	a pointer to a function that should be executed by the <code>uThread</code> .
<i>arg1</i>	first argument of the function (can be <code>nullptr</code>)
<i>arg2</i>	second argument of the function (can be <code>nullptr</code>)
<i>arg3</i>	third argument of the function (can be <code>nullptr</code>)

After creating the `uThread` and allocating the stack, the `start()` function should be called to get the `uThread` going.

5.9.3.11 `void uThread::terminate () [static]`

Terminates the `uThread`.

By calling this function `uThread` is being terminated and `uThread` object is either destroyed or put back into the cache.

5.9.3.12 `void uThread::yield () [static]`

Causes `uThread` to yield.

`uThread` give up the execution context and place itself back on the ReadyQueue of the `Cluster`. If there is no other `uThreads` available to switch to, the current `uThread` continues execution.

The documentation for this class was generated from the following files:

- `src/runtime/uThread.h`
- `src/runtime/uThread.cpp`

5.10 `uThreadCache` Class Reference

Data structure to cache `uThreads`.

```
#include <uThreadCache.h>
```

Public Member Functions

- **uThreadCache** (size_t size=defaultuThreadCacheSize)
- ssize_t **push** (uThread *ut)
adds a uThread to the cache
- uThread * **pop** ()
pop a uThread from the list in FIFO order and return it

5.10.1 Detailed Description

Data structure to cache uThreads.

uThreadCache is a linked list of uThreads using and intrusive container to cache all terminated uThreads. Instead of destroying the memory allocated for the stack, simply reset the stack pointer and push it to the cache.

5.10.2 Member Function Documentation

5.10.2.1 uThread* uThreadCache::pop () [inline]

pop a **uThread** from the list in FIFO order and return it

Returns

nullptr on failure, or a pointer to a **uThread** on success

5.10.2.2 ssize_t uThreadCache::push (uThread * ut) [inline]

adds a **uThread** to the cache

Parameters

<i>ut</i>	pointer to a uThread
-----------	-----------------------------

Returns

size of the cache if push was successful or -1 if not

This function tries to push a **uThread** into the cache structure. If the cache is full or the mutex cannot be acquired immediately the operation has failed and the function returns -1. Otherwise, it adds the **uThread** to the list and return the size of the cache.

The documentation for this class was generated from the following file:

- src/runtime/uThreadCache.h

Chapter 6

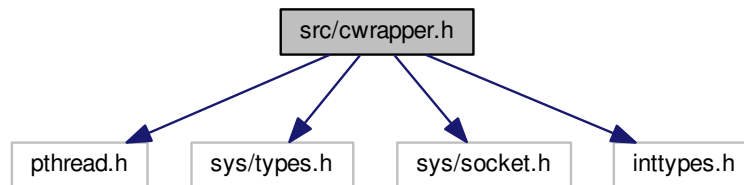
File Documentation

6.1 src/cwrapper.h File Reference

C Wrapper for uThreads.

```
#include <pthread.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <inttypes.h>
```

Include dependency graph for cwrapper.h:



Cluster

C interface for class [Cluster](#).

- typedef struct WCluster **WCluster**
- WCluster * [cluster_create](#) ()
- void [cluster_destroy](#) (WCluster *cluster)
- WCluster * [cluster_get_default](#) ()
- WCluster * [cluster_get_current](#) ()
- uint64_t [cluster_get_id](#) (WCluster *cluster)
Get the ID of [Cluster](#).
- size_t [cluster_get_number_of_kThreads](#) (WCluster *cluster)
Total number of kThreads belonging to this cluster.

kThread

C interface for class [kThread](#).

- typedef struct WkThread **WkThread**
- WkThread * [kThread_create](#) (WCluster *cluster)
- void [kThread_destroy](#) (WkThread *kt)
- uint64_t [kThread_get_total_number_of_kThreads](#) ()
- WkThread * [kThread_get_current](#) ()
Get the pointer to the current [kThread](#).
- pthread_t [kThread_get_current_pthread_id](#) ()
return the native handle for the kernel thread
- pthread_t [kThread_get_pthread_id](#) (WkThread *kt)
return the native handle for the kernel thread

uThread

C interface for class [uThread](#).

- typedef struct WuThread **WuThread**
- WuThread * [uThread_create](#) (bool joinable=false)
Create a [uThread](#) with a given stack size.
- void [uThread_start](#) (WuThread *ut, WCluster *cluster, void *func, void *arg1=0, void *arg2=0, void *arg3=0)
start the [uThread](#) by calling the function passed to it
- void [uThread_migrate](#) (WCluster *cluster)
Move the [uThread](#) to the provided cluster.
- void [uThread_terminate](#) (WuThread *ut)
Terminates the [uThread](#).
- void [uThread_yield](#) ()
Causes [uThread](#) to yield.
- bool [uThread_join](#) (WuThread *ut)
Wait for [uThread](#) to finish execution and exit.
- void [uThread_detach](#) (WuThread *ut)
Detach a joinable thread.
- uint64_t [uThread_get_id](#) (WuThread *ut)
get the ID of this [uThread](#)
- WuThread * [uThread_get_current](#) ()
Get a pointer to the current running [uThread](#).
- uint64_t [uThread_get_total_number_of_uThreads](#) ()

Connection

C interface for class [Connection](#).

- typedef struct WConnection **WConnection**
- WConnection * [connection_create](#) ()
Create a [Connection](#) that does not have.
- WConnection * [connection_create_with_fd](#) (int fd)
Create a connection object with the provided fd.
- WConnection * [connection_create_on_cluster](#) (WCluster *cluster, int fd)
Create a connection object with the provided fd, and add it to the poller thread of the provided cluster.

- WConnection * [connection_create_socket](#) (int domain, int type, int protocol)
Same as socket syscall adds | SOCK_NONBLOCK to type.
- void [connection_destory](#) (WConnection *c)
- int [connection_accept](#) (WConnection *acceptor, WConnection *conn, struct sockaddr *addr, socklen_t *addrlen)
nonblocking accept syscall and updating the passed [Connection](#) object
- WConnection * [connection_accept_connexion](#) (WConnection *acceptor, struct sockaddr *addr, socklen_t *addrlen)
Accepts a connection and returns a connection object.
- WConnection * [connection_accept_on_cluster](#) (WConnection *acceptor, WCluster *cluster, struct sockaddr *addr, socklen_t *addrlen)
Accepts a connection, adds it to the poller thread of the provided cluster, and returns a connection object.
- int [connection_socket](#) (WConnection *conn, int domain, int type, int protocol)
Same as socket syscall, set the fd for current connection.
- int [connection_listen](#) (WConnection *conn, int backlog)
Same as listen syscall on current fd.
- int [connection_bind](#) (WConnection *conn, const struct sockaddr *addr, socklen_t addrlen)
Same as bind syscall.
- int [connection_connect](#) (WConnection *conn, const struct sockaddr *addr, socklen_t addrlen)
Same as connect syscall.
- int [connection_close](#) (WConnection *conn)
closes the socket
- ssize_t [connection_recv](#) (WConnection *conn, void *buf, size_t len, int flags)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_recvfrom](#) (WConnection *conn, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_recvmsg](#) (WConnection *conn, int sockfd, struct msghdr *msg, int flags)
Call the underlying system call on [Connection](#)'s file descriptor.
- int [connection_recvmsg](#) (WConnection *conn, int sockfd, struct mmsghdr *msgvec, unsigned int vlen, unsigned int flags, struct timespec *timeout)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_send](#) (WConnection *conn, const void *buf, size_t len, int flags)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_sendto](#) (WConnection *conn, int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_sendmsg](#) (WConnection *conn, const struct msghdr *msg, int flags)
Call the underlying system call on [Connection](#)'s file descriptor.
- int [connection_sendmsg](#) (WConnection *conn, int sockfd, struct mmsghdr *msgvec, unsigned int vlen, unsigned int flags)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_read](#) (WConnection *conn, void *buf, size_t count)
Call the underlying system call on [Connection](#)'s file descriptor.
- ssize_t [connection_write](#) (WConnection *conn, const void *buf, size_t count)
Call the underlying system call on [Connection](#)'s file descriptor.
- void [connection_block_on_read](#) (WConnection *conn)
Block [uThread](#), waiting for fd to become ready for read.
- void [connection_block_on_write](#) (WConnection *conn)
Block [uThread](#), waiting for fd to become ready for write.
- int [connection_get_fd](#) (WConnection *conn)

Mutex

C interface for class [Mutex](#).

- typedef struct WMutex **WMutex**
- WMutex * [mutex_create](#) ()
- void [mutex_destroy](#) (WMutex *mutex)
- bool [mutex_acquire](#) (WMutex *mutex)
acquire the mutex
- void [mutex_release](#) (WMutex *mutex)
release the [Mutex](#)

OwnerLock

C interface for class [OwnerLock](#).

- typedef struct WOwnerLock **WOwnerLock**
- WOwnerLock * [ownerlock_create](#) ()
- void [ownerlock_destroy](#) (WOwnerLock *olock)
- uint64_t [ownerlock_acquire](#) (WOwnerLock *olock)
- void [ownerlock_release](#) (WOwnerLock *olock)

ConditionVariable

C interface for class [ConditionVariable](#).

- typedef struct WConditionVariable **WConditionVariable**
- WConditionVariable * [condition_variable_create](#) ()
- void [condition_variable_destroy](#) (WConditionVariable *cv)
- void [condition_variable_wait](#) (WConditionVariable *cv, WMutex *mutex)
Block [uThread](#) on the condition variable using the provided mutex.
- void [condition_variable_signal](#) (WConditionVariable *cv, WMutex *mutex)
Unblock one [uThread](#) waiting on the condition variable.
- void [condition_variable_signal_all](#) (WConditionVariable *cv, WMutex *mutex)
unblock all [uThreads](#) waiting on the condition variable
- bool [condition_variable_empty](#) (WConditionVariable *cv)
Whether the waiting list is empty or not.

Semaphore

C interface for class [Semaphore](#).

- typedef struct WSemaphore **WSemaphore**
- WSemaphore * [semaphore_create](#) ()
Create a new [Semaphore](#).
- void [semaphore_destroy](#) (WSemaphore *sem)
- bool [semaphore_p](#) (WSemaphore *sem)
Decrement the value of the [Semaphore](#).
- void [semaphore_v](#) (WSemaphore *sem)
increment the value of the [Semaphore](#)

uThreadPool

C interface for class uThreadPool.

- typedef struct WuThreadPool **WuThreadPool**
- WuThreadPool * **uthreadpool_create** ()
- void **uthreadpool_destory** (WuThreadPool *utp)
- void **uthreadpool_execute** (WuThreadPool *utp, WCluster *cluster, void *(*start_routine)(void *), void *arg)

6.1.1 Detailed Description

C Wrapper for uThreads.

Author

Saman Barghi

6.1.2 Function Documentation

6.1.2.1 WCluster* cluster_create ()

Create a new [Cluster](#)

6.1.2.2 void cluster_destroy (WCluster * cluster)

Destroy the cluster

6.1.2.3 WCluster* cluster_get_current ()

6.1.2.4 WCluster* cluster_get_default ()

Returns

defaultCluster

6.1.2.5 uint64_t cluster_get_id (WCluster * cluster)

Get the ID of [Cluster](#).

Returns

The ID of the cluster

6.1.2.6 size_t cluster_get_number_of_kThreads (WCluster * cluster)

Total number of kThreads belonging to this cluster.

Returns

Total number of kThreads belonging to this cluster

6.1.2.7 `WConditionVariable* condition_variable_create ()`

6.1.2.8 `void condition_variable_destroy (WConditionVariable * cv)`

6.1.2.9 `bool condition_variable_empty (WConditionVariable * cv)`

Whether the waiting list is empty or not.

Returns

Whether the waiting list is empty or not

6.1.2.10 `void condition_variable_signal (WConditionVariable * cv, WMutex * mutex)`

Unblock one [uThread](#) waiting on the condition variable.

Parameters

<i>mutex</i>	The mutex to be released after unblocking is done
--------------	---

6.1.2.11 `void condition_variable_signal_all (WConditionVariable * cv, WMutex * mutex)`

unblock all [uThreads](#) waiting on the condition variable

Parameters

<i>mutex</i>	The mutex to be released after unblocking is done
--------------	---

6.1.2.12 `void condition_variable_wait (WConditionVariable * cv, WMutex * mutex)`

Block [uThread](#) on the condition variable using the provided mutex.

Parameters

<i>mutex</i>	used to synchronize access to the condition
--------------	---

6.1.2.13 `int connection_accept (WConnection * acceptor, WConnection * conn, struct sockaddr * addr, socklen_t * addrlen)`

nonblocking accept syscall and updating the passed [Connection](#) object

Parameters

<i>conn</i>	Pointer to a Connection object that is not initialized
-------------	--

Returns

same as accept system call

This format is used for compatibility with C

6.1.2.14 `WConnection* connection_accept_connenction (WConnection * acceptor, struct sockaddr * addr, socklen_t * addrlen)`

Accepts a connection and returns a connection object.

Returns

Newly created connection

Throws a `std::system_error` exception on error. Never call from C.

6.1.2.15 `WConnection* connection_accept_on_cluster (WConnection * acceptor, WCluster * cluster, struct sockaddr * addr, socklen_t * addrlen)`

Accepts a connection, adds it to the poller thread of the provided cluster, and returns a connection object.

Returns

Newly created connection

Throws a `std::system_error` exception on error. Never call from C.

6.1.2.16 `int connection_bind (WConnection * conn, const struct sockaddr * addr, socklen_t addrlen)`

Same as `bind` syscall.

Returns

Same as `bind` syscall

6.1.2.17 `void connection_block_on_read (WConnection * conn)`

Block [uThread](#), waiting for fd to become ready for read.

6.1.2.18 `void connection_block_on_write (WConnection * conn)`

Block [uThread](#), waiting for fd to become ready for write.

6.1.2.19 `int connection_close (WConnection * conn)`

closes the socket

Returns

the same as `close` system call

6.1.2.20 `int connection_connect (WConnection * conn, const struct sockaddr * addr, socklen_t addrlen)`

Same as `connect` syscall.

Returns

Same as `connect` syscall

6.1.2.21 `WConnection* connection_create ()`

Create a [Connection](#) that does not have.

This is useful for `accept` or `socket` functions that require a [Connection](#) object without fd being set

6.1.2.22 `WConnection* connection_create_on_cluster (WCluster * cluster, int fd)`

Create a connection object with the provided fd, and add it to the poller thread of the provided cluster.

Parameters

<i>fd</i>	If the connection is already established by other means, set the fd and add it to the polling structure
-----------	---

6.1.2.23 `WConnection* connection_create_socket (int domain, int type, int protocol)`

Same as socket syscall adds | SOCK_NONBLOCK to type.

Returns

same as socket syscall

Throws a `std::system_error` exception. Do not call from C code. The unerlying socket is always nonblocking. This is achieved by adding a (| SOCK_NONBLOCK) to type, thus requires linux kernels > 2.6.27

6.1.2.24 `WConnection* connection_create_with_fd (int fd)`

Create a connection object with the provided fd.

Parameters

<i>fd</i>	If the connection is already established by other means, set the fd and add it to the polling structure
-----------	---

6.1.2.25 `void connection_destory (WConnection * c)`

6.1.2.26 `int connection_get_fd (WConnection * conn)`

6.1.2.27 `int connection_listen (WConnection * conn, int backlog)`

Same as listen syscall on current fd.

Returns

Same as listen syscall

6.1.2.28 `ssize_t connection_read (WConnection * conn, void * buf, size_t count)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.29 `ssize_t connection_recv (WConnection * conn, void * buf, size_t len, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.30 `ssize_t connection_recvfrom (WConnection * conn, void * buf, size_t len, int flags, struct sockaddr * src_addr, socklen_t * addrlen)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.31 `int connection_recvmsg (WConnection * conn, int sockfd, struct mmsghdr * msgvec, unsigned int vlen, unsigned int flags, struct timespec * timeout)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.32 `ssize_t connection_recvmsg (WConnection * conn, int sockfd, struct msghdr * msg, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.33 `ssize_t connection_send (WConnection * conn, const void * buf, size_t len, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.34 `int connection_sendmmsg (WConnection * conn, int sockfd, struct mmsghdr * msgvec, unsigned int vlen, unsigned int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.35 `ssize_t connection_sendmsg (WConnection * conn, const struct msghdr * msg, int flags)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.36 `ssize_t connection_sendto (WConnection * conn, int sockfd, const void * buf, size_t len, int flags, const struct sockaddr * dest_addr, socklen_t addrlen)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.37 `int connection_socket (WConnection * conn, int domain, int type, int protocol)`

Same as socket syscall, set the fd for current connection.

Returns

same as socket syscall The underlying socket is always nonblocking. This is achieved by adding a (`| SOCK_NONBLOCK`) to type, thus requires linux kernels `> 2.6.27`

6.1.2.38 `ssize_t connection_write (WConnection * conn, const void * buf, size_t count)`

Call the underlying system call on [Connection](#)'s file descriptor.

Returns

Same as what the related syscall returns

This function calls the system call with the same name. If the socket is ready for the required function it returns immediately, otherwise it blocks in the user-level (blocks [uThread](#) not [kThread](#)), and polls the file descriptor until it becomes ready.

The return results is the same as the underlying system call except that the following condition is never true when the function returns: `(res == -1) && (errno == EAGAIN || errno == EWOULDBLOCK)`

which means the [Connection](#) object does the polling and only returns when an error occurs or the socket is ready.

6.1.2.39 `WkThread* kThread_create (WCluster * cluster)`

6.1.2.40 `void kThread_destroy (WkThread * kt)`

6.1.2.41 `WkThread* kThread_get_current ()`

Get the pointer to the current [kThread](#).

Returns

current [kThread](#)

This is necessary when a [uThread](#) wants to find which [kThread](#) it is being executed over.

6.1.2.42 pthread_t kThread_get_current_pthread_id ()

return the native handle for the kernel thread

Returns

native handle for the [kThread](#)

In linux this is pthread_t representation of the thread.

6.1.2.43 pthread_t kThread_get_pthread_id (WkThread * kt)

return the native handle for the kernel thread

Returns

native handle for the [kThread](#)

In linux this is pthread_t representation of the thread.

6.1.2.44 uint64_t kThread_get_total_number_of_kThreads ()

Returns

total number of kThreads running under the program.

6.1.2.45 bool mutex_acquire (WMutex * mutex)

acquire the mutex

Returns

true if it was acquired, false otherwise

The return value is only for when timeouts are implemented

6.1.2.46 WMutex* mutex_create ()

6.1.2.47 void mutex_destroy (WMutex * mutex)

6.1.2.48 void mutex_release (WMutex * mutex)

release the [Mutex](#)

6.1.2.49 uint64_t ownerlock_acquire (WOwnerLock * olock)

6.1.2.50 WOwnerLock* ownerlock_create ()

6.1.2.51 void ownerlock_destroy (WOwnerLock * olock)

6.1.2.52 void ownerlock_release (WOwnerLock * olock)

6.1.2.53 WSemaphore* semaphore_create ()

Create a new [Semaphore](#).

Parameters

<i>c</i>	Initial value of the Semaphore
----------	--

6.1.2.54 `void semaphore_destroy (WSemaphore * sem)`

6.1.2.55 `bool semaphore_p (WSemaphore * sem)`

Decrement the value of the [Semaphore](#).

Returns

Whether it was successful or not

6.1.2.56 `void semaphore_v (WSemaphore * sem)`

increment the value of the [Semaphore](#)

6.1.2.57 `WuThread* uThread_create (bool joinable = false)`

Create a [uThread](#) with a given stack size.

Parameters

<i>ss</i>	stack size
<i>joinable</i>	Whether this thread is joinable or detached

Returns

a pointer to a new [uThread](#)

This function relies on a [uThreadCache](#) structure and does not always allocate the stack.

6.1.2.58 `void uThread_detach (WuThread * ut)`

Detach a joinable thread.

6.1.2.59 `WuThread* uThread_get_current ()`

Get a pointer to the current running [uThread](#).

Returns

pointer to the current [uThread](#)

6.1.2.60 `uint64_t uThread_get_id (WuThread * ut)`

get the ID of this [uThread](#)

Returns

ID of the [uThread](#)

6.1.2.61 `uint64_t uThread_get_total_number_of_uThreads ()`

Returns

Total number of uThreads in the program

This number does not include mainUT or IOUs

6.1.2.62 `bool uThread_join (WuThread * ut)`

Wait for [uThread](#) to finish execution and exit.

Returns

Whether join was successful or failed

6.1.2.63 `void uThread_migrate (WCluster * cluster)`

Move the [uThread](#) to the provided cluster.

Parameters

<i>cluster</i>	This function is used to migrate the uThread to another Cluster . Migration is useful specially if clusters form a pipeline of execution.
----------------	---

6.1.2.64 `void uThread_start (WuThread * ut, WCluster * cluster, void * func, void * arg1 = 0, void * arg2 = 0, void * arg3 = 0)`

start the [uThread](#) by calling the function passed to it

Parameters

<i>cluster</i>	The cluster that function belongs to.
<i>func</i>	a pointer to a function that should be executed by the uThread .
<i>arg1</i>	first argument of the function (can be nullptr)
<i>arg2</i>	second argument of the function (can be nullptr)
<i>arg3</i>	third argument of the function (can be nullptr)

After creating the [uThread](#) and allocating the stack, the start() function should be called to get the [uThread](#) going.

6.1.2.65 `void uThread_terminate (WuThread * ut)`

Terminates the [uThread](#).

By calling this function [uThread](#) is being terminated and [uThread](#) object is either destroyed or put back into the cache.

6.1.2.66 `void uThread_yield ()`

Causes [uThread](#) to yield.

[uThread](#) give up the execution context and place itself back on the ReadyQueue of the [Cluster](#). If there is no other uThreads available to switch to, the current [uThread](#) continues execution.

Index

- ~Cluster
 - Cluster, [14](#)
- accept
 - Connection, [18](#), [19](#)
- acquire
 - Mutex, [27](#)
 - OwnerLock, [28](#)
- bind
 - Connection, [19](#)
- BlockingQueue, [11](#)
 - signal, [12](#)
 - signalAll, [12](#)
 - suspend, [13](#)
- close
 - Connection, [19](#)
- Cluster, [13](#)
 - ~Cluster, [14](#)
 - Cluster, [14](#)
 - getDefaultCluster, [14](#)
 - getID, [14](#)
 - getNumberOfkThreads, [15](#)
 - operator=, [15](#)
- cluster_create
 - cwrapper.h, [41](#)
- cluster_destroy
 - cwrapper.h, [41](#)
- cluster_get_current
 - cwrapper.h, [41](#)
- cluster_get_default
 - cwrapper.h, [41](#)
- cluster_get_id
 - cwrapper.h, [41](#)
- cluster_get_number_of_kThreads
 - cwrapper.h, [41](#)
- condition_variable_create
 - cwrapper.h, [41](#)
- condition_variable_destroy
 - cwrapper.h, [42](#)
- condition_variable_empty
 - cwrapper.h, [42](#)
- condition_variable_signal
 - cwrapper.h, [42](#)
- condition_variable_signal_all
 - cwrapper.h, [42](#)
- condition_variable_wait
 - cwrapper.h, [42](#)
- ConditionVariable, [15](#)
- empty, [16](#)
- signal, [16](#)
- signalAll, [16](#)
- wait, [16](#)
- connect
 - Connection, [19](#)
- Connection, [16](#)
 - accept, [18](#), [19](#)
 - bind, [19](#)
 - close, [19](#)
 - connect, [19](#)
 - Connection, [18](#)
 - getFd, [19](#)
 - listen, [19](#)
 - read, [19](#)
 - recv, [20](#)
 - recvfrom, [20](#)
 - recvmsg, [20](#)
 - recvmsg, [21](#)
 - send, [21](#)
 - sendmsg, [21](#)
 - sendmsg, [21](#)
 - sendto, [22](#)
 - socket, [22](#)
 - write, [22](#)
- connection_accept
 - cwrapper.h, [42](#)
- connection_accept_connenction
 - cwrapper.h, [42](#)
- connection_accept_on_cluster
 - cwrapper.h, [43](#)
- connection_bind
 - cwrapper.h, [43](#)
- connection_block_on_read
 - cwrapper.h, [43](#)
- connection_block_on_write
 - cwrapper.h, [43](#)
- connection_close
 - cwrapper.h, [43](#)
- connection_connect
 - cwrapper.h, [43](#)
- connection_create
 - cwrapper.h, [43](#)
- connection_create_on_cluster
 - cwrapper.h, [43](#)
- connection_create_socket
 - cwrapper.h, [45](#)
- connection_create_with_fd
 - cwrapper.h, [45](#)

- connection_destory
 - cwrapper.h, 45
- connection_get_fd
 - cwrapper.h, 45
- connection_listen
 - cwrapper.h, 45
- connection_read
 - cwrapper.h, 45
- connection_recv
 - cwrapper.h, 45
- connection_recvfrom
 - cwrapper.h, 46
- connection_recvmmsg
 - cwrapper.h, 46
- connection_recvmsg
 - cwrapper.h, 46
- connection_send
 - cwrapper.h, 47
- connection_sendmmsg
 - cwrapper.h, 47
- connection_sendmsg
 - cwrapper.h, 47
- connection_sendto
 - cwrapper.h, 47
- connection_socket
 - cwrapper.h, 48
- connection_write
 - cwrapper.h, 48
- create
 - uThread, 32, 33
- currentUThread
 - uThread, 33
- currentkThread
 - kThread, 24
- cwrapper.h
 - cluster_create, 41
 - cluster_destroy, 41
 - cluster_get_current, 41
 - cluster_get_default, 41
 - cluster_get_id, 41
 - cluster_get_number_of_kThreads, 41
 - condition_variable_create, 41
 - condition_variable_destroy, 42
 - condition_variable_empty, 42
 - condition_variable_signal, 42
 - condition_variable_signal_all, 42
 - condition_variable_wait, 42
 - connection_accept, 42
 - connection_accept_connenction, 42
 - connection_accept_on_cluster, 43
 - connection_bind, 43
 - connection_block_on_read, 43
 - connection_block_on_write, 43
 - connection_close, 43
 - connection_connect, 43
 - connection_create, 43
 - connection_create_on_cluster, 43
 - connection_create_socket, 45
 - connection_create_with_fd, 45
 - connection_destory, 45
 - connection_get_fd, 45
 - connection_listen, 45
 - connection_read, 45
 - connection_recv, 45
 - connection_recvfrom, 46
 - connection_recvmmsg, 46
 - connection_recvmsg, 46
 - connection_send, 47
 - connection_sendmmsg, 47
 - connection_sendmsg, 47
 - connection_sendto, 47
 - connection_socket, 48
 - connection_write, 48
 - kThread_create, 48
 - kThread_destroy, 48
 - kThread_get_current, 48
 - kThread_get_current_pthread_id, 48
 - kThread_get_pthread_id, 49
 - kThread_get_total_number_of_kThreads, 49
 - mutex_acquire, 49
 - mutex_create, 49
 - mutex_destroy, 49
 - mutex_release, 49
 - ownerlock_acquire, 49
 - ownerlock_create, 49
 - ownerlock_destroy, 49
 - ownerlock_release, 49
 - semaphore_create, 49
 - semaphore_destroy, 50
 - semaphore_p, 50
 - semaphore_v, 50
 - uThread_create, 50
 - uThread_detach, 50
 - uThread_get_current, 50
 - uThread_get_id, 50
 - uThread_get_total_number_of_uThreads, 50
 - uThread_join, 51
 - uThread_migrate, 51
 - uThread_start, 51
 - uThread_terminate, 51
 - uThread_yield, 51
- empty
 - ConditionVariable, 16
- getCurrentCluster
 - uThread, 33
- getDefaultCluster
 - Cluster, 14
- getFd
 - Connection, 19
- getID
 - Cluster, 14
 - kThread, 24
 - uThread, 33
- getNumberOfkThreads
 - Cluster, 15

- getThreadNativeHandle
 - kThread, [25](#)
- getTotalNumberOfkThreads
 - kThread, [25](#)
- getTotalNumberOfUTs
 - uThread, [33](#)
- join
 - uThread, [33](#)
- kThread, [23](#)
 - currentkThread, [24](#)
 - getID, [24](#)
 - getThreadNativeHandle, [25](#)
 - getTotalNumberOfkThreads, [25](#)
 - kThread, [24](#)
 - kThread, [24](#)
 - operator=, [25](#)
- kThread_create
 - cwrapper.h, [48](#)
- kThread_destroy
 - cwrapper.h, [48](#)
- kThread_get_current
 - cwrapper.h, [48](#)
- kThread_get_current_pthread_id
 - cwrapper.h, [48](#)
- kThread_get_pthread_id
 - cwrapper.h, [49](#)
- kThread_get_total_number_of_kThreads
 - cwrapper.h, [49](#)
- listen
 - Connection, [19](#)
- migrate
 - uThread, [34](#)
- Mutex, [25](#)
 - acquire, [27](#)
- mutex_acquire
 - cwrapper.h, [49](#)
- mutex_create
 - cwrapper.h, [49](#)
- mutex_destroy
 - cwrapper.h, [49](#)
- mutex_release
 - cwrapper.h, [49](#)
- operator=
 - Cluster, [15](#)
 - kThread, [25](#)
 - uThread, [34](#)
- OwnerLock, [27](#)
 - acquire, [28](#)
 - release, [28](#)
- ownerlock_acquire
 - cwrapper.h, [49](#)
- ownerlock_create
 - cwrapper.h, [49](#)
- ownerlock_destroy
 - cwrapper.h, [49](#)
- ownerlock_release
 - cwrapper.h, [49](#)
- P
 - Semaphore, [29](#)
- pop
 - uThreadCache, [35](#)
- push
 - uThreadCache, [35](#)
- read
 - Connection, [19](#)
- recv
 - Connection, [20](#)
- recvfrom
 - Connection, [20](#)
- recvmsg
 - Connection, [20](#)
- recvmsg
 - Connection, [21](#)
- release
 - OwnerLock, [28](#)
- Semaphore, [29](#)
 - P, [29](#)
 - Semaphore, [29](#)
- semaphore_create
 - cwrapper.h, [49](#)
- semaphore_destroy
 - cwrapper.h, [50](#)
- semaphore_p
 - cwrapper.h, [50](#)
- semaphore_v
 - cwrapper.h, [50](#)
- send
 - Connection, [21](#)
- sendmsg
 - Connection, [21](#)
- sendmsg
 - Connection, [21](#)
- sendto
 - Connection, [22](#)
- signal
 - BlockingQueue, [12](#)
 - ConditionVariable, [16](#)
- signalAll
 - BlockingQueue, [12](#)
 - ConditionVariable, [16](#)
- socket
 - Connection, [22](#)
- src/cwrapper.h, [37](#)
- start
 - uThread, [34](#)
- suspend
 - BlockingQueue, [13](#)
- terminate
 - uThread, [34](#)

- uThread, [30](#)
 - create, [32](#), [33](#)
 - currentUThread, [33](#)
 - getCurrentCluster, [33](#)
 - getID, [33](#)
 - getTotalNumberOfUTs, [33](#)
 - join, [33](#)
 - migrate, [34](#)
 - operator=, [34](#)
 - start, [34](#)
 - terminate, [34](#)
 - uThread, [32](#)
 - uThread, [32](#)
 - yield, [34](#)
- uThread_create
 - cwrapper.h, [50](#)
- uThread_detach
 - cwrapper.h, [50](#)
- uThread_get_current
 - cwrapper.h, [50](#)
- uThread_get_id
 - cwrapper.h, [50](#)
- uThread_get_total_number_of_uThreads
 - cwrapper.h, [50](#)
- uThread_join
 - cwrapper.h, [51](#)
- uThread_migrate
 - cwrapper.h, [51](#)
- uThread_start
 - cwrapper.h, [51](#)
- uThread_terminate
 - cwrapper.h, [51](#)
- uThread_yield
 - cwrapper.h, [51](#)
- uThreadCache, [34](#)
 - pop, [35](#)
 - push, [35](#)
- wait
 - ConditionVariable, [16](#)
- write
 - Connection, [22](#)
- yield
 - uThread, [34](#)