uThreads

0.1.0

Generated by Doxygen 1.8.6

Tue Feb 23 2016 14:48:36

Contents

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:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

Class Documentation

4.1 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

Friends

- class **uThread**
- class **Mutex**
- class **OwnerLock**
- class **ConditionVariable**
- class **Semaphore**

4.1.1 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.

4.1.2 Member Function Documentation

4.1.2.1 bool BlockingQueue::signal (std::mutex & *lock,* uThread ∗& *owner*)

Unblock one blocked uThread, used for OwnerLock.

Parameters

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

Returns

true if a uThread was unblocked, and false otherwise

4.1.2.2 bool BlockingQueue::signal (std::mutex & *lock*) [inline]

unblock one blocked, used for Mutex

Parameters

lock	mutex to be released after signal is done

Returns

true if a uThread was unblocked, and false otherwise

4.1.2.3 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

4.1.2.4 void BlockingQueue::signalAll (Mutex & *mutex*)

unblock all blocked uThreads, used for Condition Variable

Parameters

Mutex	to be released after signalAll is done

4.1.2.5 bool BlockingQueue::suspend (std::mutex & *lock*)

Suspends the uThread and add it to the queue.

Parameters

lock	a mutex to be released after blocking

Returns

whether the suspension was successful or not

Suspends the uThread and adds it to the BlockingQueue.

4.1.2.6 bool BlockingQueue::suspend (**Mutex** & *mutex*)

Suspends the uThread and add it to the queue.

Parameters

lock	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

4.2 Cluster Class Reference

Scheduler and Cluster of kThreads.

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

Public Member Functions

- 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.

Static Public Member Functions

- static Cluster & getDefaultCluster ()

Friends

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

4.2.1 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.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 Cluster::Cluster ( )

Create a new Cluster

4.2.2.2 Cluster::Cluster (const Cluster &) [delete]

Cluster cannot be copied or assigned.

4.2.3 Member Function Documentation

4.2.3.1 static Cluster& Cluster::getDefaultCluster ( ) [inline], [static]

Returns

defaultCluster

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

Get the ID of Cluster.

Returns

The ID of the cluster

4.2.3.3 size_t Cluster::getNumberOfkThreads ( ) const [inline]

Total number of kThreads belonging to this cluster.

Returns

Total number of kThreads belonging to this cluster

4.2.3.4 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

4.3 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.

4.3.1 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.

4.3.2 Member Function Documentation

4.3.2.1 bool ConditionVariable::empty ( ) [inline]

Whether the waiting list is empty or not.

Returns

Whether the waiting list is empty or not

4.3.2.2 void ConditionVariable::signal (Mutex & *mutex*) [inline]

Unblock one uThread waiting on the condition variable.

Parameters

mutex	The mutex to be released after unblocking is done

4.3.2.3 void ConditionVariable::signalAll (**Mutex & *mutex***) [inline]

unblock all uThreads waiting on the condition variable

Parameters

mutex	The mutex to be released after unblocking is done

4.3.2.4 void ConditionVariable::wait (**Mutex & *mutex***) [inline]

Block uThread on the condition variable using the provided mutex.

Parameters

mutex	used to synchronize access to the condition

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

- src/runtime/BlockingSync.h

4.4 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 recvmmsg (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 sendmmsg (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 read to happen.
- void blockOnWrite ()

  Block uThread, waiting for write to happen.
- int close ()

  closes the socket
- int getFd () const

  return the Connection's file descriptor

4.4.1 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.

4.4.2 Constructor & Destructor Documentation

4.4.2.1 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

4.4.2.2 Connection::Connection (int *fd*) [inline]

Create a connection object with the provided fd.

Parameters

fd	If the connection is already established by other means, set the fd and add it to the polling structure

4.4.2.3 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

fd	If the connection is already established by other means, set the fd and add it to the polling structure

4.4.2.4 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 nonbelocking. This is achieved by adding a (| SOCK_NONBLOCK) to type, thus requires linux kernels > 2.6.27

4.4.3 Member Function Documentation

4.4.3.1 int Connection::accept (Connection ∗ *conn,* struct sockaddr ∗ *addr,* socklen_t ∗ *addrlen*)

nonblocking accept syscall and updating the passed Connection object

Parameters

conn	Pointer to a Connection object that is not initialized

Returns

same as accept system call

This format is used for compatibility with C

4.4.3.2 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.

4.4.3.3 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.

4.4.3.4 int Connection::bind (const struct sockaddr ∗ *addr,* socklen_t *addrlen*)

Same as bind syscall.

Returns

Same as bind syscall

4.4.3.5 int Connection::close ( )

closes the socket

Returns

the same as close system call

4.4.3.6 int Connection::connect (const struct sockaddr ∗ *addr,* socklen_t *addrlen*)

Same as connect syscall.

Returns

Same as connect syscall

4.4.3.7 int Connection::getFd ( ) const [inline]

return the Connection's file descriptor

Returns

file descriptor

4.4.3.8 int Connection::listen (int *backlog*)

Same as listen syscall on current fd.

Returns

Same as listen syscall

4.4.3.9 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 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.

4.4.3.10 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 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.

4.4.3.11 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 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.

4.4.3.12 int Connection::recvmmsg (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.

4.4.3.13 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.

4.4.3.14 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.

4.4.3.15 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.

4.4.3.16 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 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.

4.4.3.17 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 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.

4.4.3.18 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 unerlying socket is always nonbelocking. This is achieved by adding a (| SOCK_-NONBLOCK) to type, thus requires linux kernels > 2.6.27

4.4.3.19 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 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.

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

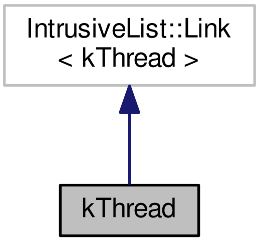
- src/io/Network.h
- src/io/Network.cpp

4.5 kThread Class Reference

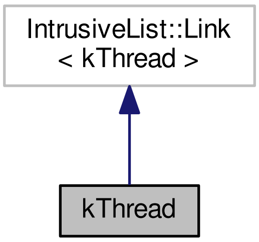
Object to represent kernel threads.

```
#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 hanlde 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**

4.5.1 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.

4.5.2 Constructor & Destructor Documentation

4.5.2.1 kThread::kThread (**Cluster** & *cluster*)

Create a kThread on the passed cluster.

Parameters

The	Cluster this kThread belongs to.

4.5.3 Member Function Documentation

4.5.3.1 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.

4.5.3.2 std::thread::id kThread::getID ( )

returns the kernel thread ID

Returns

the kThread ID

The returned type depends on the platform.

4.5.3.3 std::thread::native_handle_type kThread::getThreadNativeHandle ( )

return the native hanlde for the kernel thread

Returns

native handle for the kThread

In linux this is pthread_t representation of the thread.

4.5.3.4 static uint kThread::getTotalNumberOfkThreads ( ) [inline], [static]

Returns

total number of kThreads running under the program.

4.5.3.5 const kThread& kThread::operator= (const kThread &) [delete]

kThread cannot be copied or assigned.

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

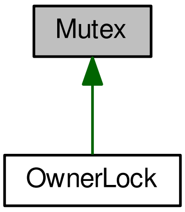
- src/runtime/kThread.h
- src/runtime/kThread.cpp
- src/runtime/uThread.cpp

4.6 Mutex Class Reference

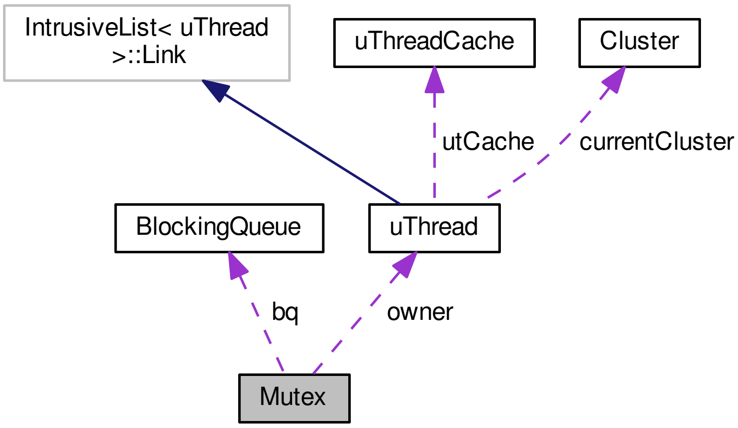
A user-level Mutex.

`#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

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**

4.6.1 Detailed Description

A user-level Mutex.

4.6.2 Member Function Documentation

4.6.2.1 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:

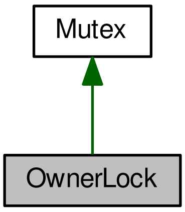
- src/runtime/BlockingSync.h

4.7 OwnerLock Class Reference

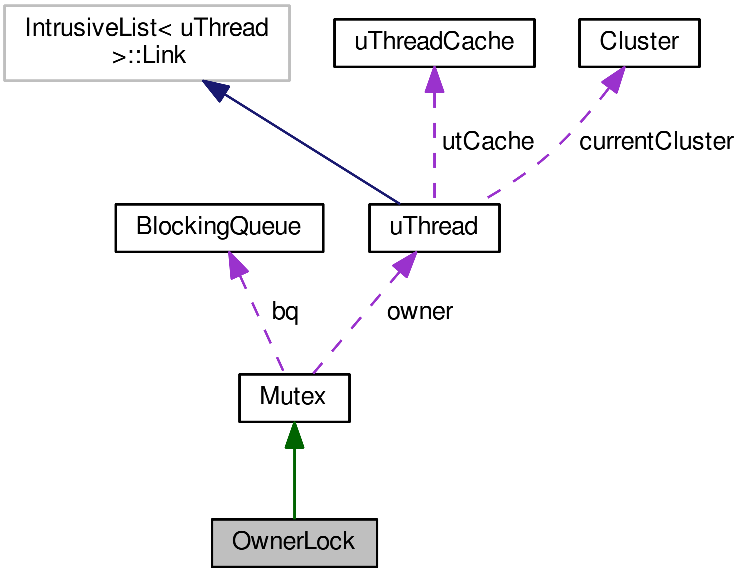
an Owner Mutex where owner can recursively acquire the Mutex

```
#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

4.7.1 Detailed Description

an Owner Mutex where owner can recursively acquire the Mutex

4.7.2 Member Function Documentation

4.7.2.1 mword OwnerLock::acquire ( ) [inline]

Acquire the OwnerLock.

Returns

The number of times current owner acquired the lock

4.7.2.2 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

4.8 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

4.8.1 Detailed Description

A user-level Semaphore.

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

4.8.2 Constructor & Destructor Documentation

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

Create a new Semaphore.

Parameters

c	Initial value of the Semaphore

4.8.3 Member Function Documentation

4.8.3.1 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:

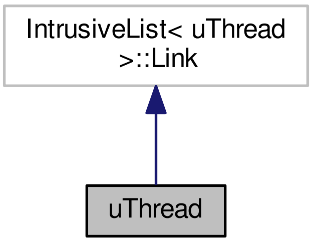
- src/runtime/BlockingSync.h

4.9 uThread Class Reference

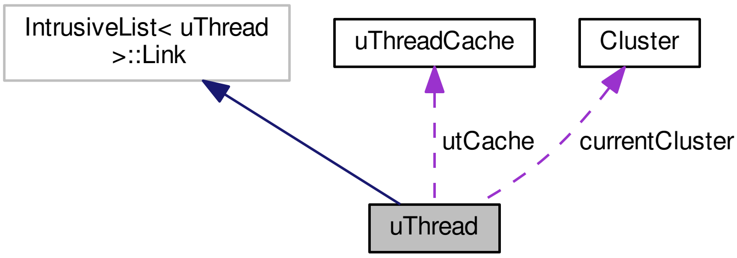
user-level threads (fiber)

`#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.
- const 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** (std::function< 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**

4.9.1 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.

4.9.2 Constructor & Destructor Documentation

4.9.2.1 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.

4.9.3 Member Function Documentation

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

Create a uThread with a given stack size.

Parameters

ss	stack size

joinable	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.

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

Create a uThread with default stack size.

Parameters

joinable	Whether this thread is joinable or detached

Returns

a pointer to a new uThread

4.9.3.3 uThread ∗ uThread::currentUThread ( ) `[static]`

Get a pointer to the current running uThread.

Returns

pointer to the current uThread

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

return the current Cluster uThread is executed on

Returns

the current Cluster uThread is executed on

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

get the ID of this uThread

Returns

ID of the uThread

4.9.3.6 static uint64_t uThread::getTotalNumberofUTs ( ) `[inline]`, `[static]`

Returns

Total number of uThreads in the program

This number does not include mainUT or IOUTs

4.9.3.7 bool uThread::join ( )

Wait for uThread to finish execution and exit.

Returns

Whether join was successful or failed

4.9.3.8 void uThread::migrate (Cluster ∗ *cluster*) [static]

Move the uThread to the provided cluster.

Parameters

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

4.9.3.9 const uThread& uThread::operator= (const uThread &) [delete]

4.9.3.10 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

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

4.9.3.11 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.

4.9.3.12 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

4.10 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

4.10.1 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.

4.10.2 Member Function Documentation

4.10.2.1 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

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

adds a uThread to the cache

Parameters

ut	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

Index