uThreads

0.3.0

Generated by Doxygen 1.8.11

Contents

Chapter 1

uThreads: Concurrent User Threads in C++(and C)

What are uThreads?

uThreads is a concurrent library based on cooperative scheduling of user-level threads(fibers) 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. For example, `Facebook's folly::fiber`, `libmill`, and `libtask` use N:1 mapping. Having N:1 mapping means if the application blocks at the kernel level, all user-level threads are blocked and application cannot move forward. One way to address this is to only block on user level, hence, blocking user-level threads. This setting works very well with IO bound applications, however, if a user thread requires using a CPU for a while, it can block other user threads and the task is better to be executed asynchronously on another core to prevent this from happening. In order to avoid this problem, user threads can be mapped to multiple kernel-level threads. Thus, creating the third scenario with M:N or hybrid mapping. e.g., `go` and `uC++` use 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 using a single ReadyQueue Scheduler. You can find the documentation here `http://samanbarghi.github.io/uThreads`

Webserver throughput results

Here are a comparsion of a simple webserver throughput with `fasthttp`, `cppsp`, and nodejs. Experiments are executed on my laptop (i7 quad core). Note that not much optimization is applied to any of the applications, thus there might me some space to squeeze more throughput out of these applications. You can check the source code of the sample webserver under the test directory.

Dependencies

Currently uThreads only supports Linux on x86_64 platforms. It also depends on the following:

- gcc > 4.8
- linux kernel >= 2.9

Building and installation

To build and install the library issue the following commands:

```
make
make install
```

Currently, everything is installed under /usr/local/lib and /usr/local/include/uThreads. To change this you need to edit the Makefile and change the DEST_DIR.

Usage

- Include "uThreads/uThreads.h" in your source file.
- Link your program with uThreads library (-luThreads) at compile time.

There are sample applications under *test* directory, to compile them issue `make test`, and you can find the binaries under *bin* directory. Refer to the `documentation` for more information.

uThreads structure

This section explains the internals of uThreads.

Basics

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 *Scheduler* which is used to schedule uThreads over kThreads in that Cluster. Application programmer decides how many kThreads belong to a *Cluster* by assigning them upon creation. *Clusters* can be used to execute different tasks over separate kThreads and if pinned properly, over separate cores. For exampl,e they can be used to provide better CPU cache locality for different set of tasks, by executing them on specific cores.

kThreads are kernel-level threads (std::thread), that are the main vehicle to utilize cores and execute the program. Each kThread can interact with the local scheduler in the *Cluster* and execute the *uThreads* provided by the local *Scheduler*, but it can move uThreads to another 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 uThread continue execution 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 a readyQueue or runQueue 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 *defaultk←Thread*, and also a uThread called *mainUT* to take over the defaultkThread stack and run the *main* function.

In addition, there is a single *Poller kThread* which is responsible for polling the network devices, and multiplexing network events over the Cluster. uThreads provides a user-level blocking on network events, where network calls are non-blocking at the kernel-level but uThreads block on network events if the device is not ready. The poller thread is thus responsible for unblocking the uThreads upon receiving the related network event. The poller thread is using *edge triggered epoll* in Linux, and which is similar to how `Golang` supports multiplexing of network events.

Currently, uThreads only supports **fixed** stack sizes for performance purposes. uThread's stack is cached after finishing execution to avoid the extra overhead of memory allocation.

Scheduler

As Explained earlier, each *Cluster* has a local *Scheduler* which is responsible for distributing *uThreads* among the *kThreads* whithin that *Cluster*. Currently, there are 4 different schedulers, which will be explained below. These schedulers do not support any work sharing or work stealing at the moment and based on the type of the Scheduler either uThreads are assigned in a round robin manner, or kThreads ask the Scheduler for more uThreads when they run out of work. The type of the scheduler is determined at compile time by defining **SCHEDULERNO** and pass the related scheduler number. The default scheduler is Scheduler #2, with local instrusive Multiple-Producer-Single-Consumer Queues per kThread (since it provides better performance and scalability).

- **Global ReadyQueue per *Cluster***: The first scheduler is implemented using an unbounded intrusive Ready←Queue per Cluster, and C++ synchronization primitievs (std::mutex and std::condition_variable) are used to orchestrate kThreads to access the ReadyQueue. To avoid the high overhead of mutex and condition_variable (in linux: pthread_mutex and pthread_cond) under contention, a local queue is added to each kThread, so everytime a kThread runs out of work, it simply removes many uThreads, based on the size of the Ready←Queue and the number of kThreads in that Cluster, instead of one. The local queue is only accessed by a single kThread and thus does not require mutual exclusion. The following shows the design: To measure the performance of different schedulers, the follwing experiment is designed:
- The experiment starts with 2 Clusters.
- Thre are *k* kThreads per Cluster, and the number of kThreads are changed (x-axis of the following graphs shows the number of kThreads).
- There are *1, 000, 000 × k* uThreads created when the experiment starts.
- Each uThread migrates back and forth between Clusters.
- uThread exits after 10 migrations.
- We measure the number of migrations/second and plot it on y-axis, thus higher is better.

Figure 3 shows result for the first Scheduler:

Based on the results, this approach is not very scalable past 4 kThreads per Cluster.

- **Local RunQueue per kThread using mutex and cv**: To provide better scalability, we can remove the global ReadyQueue to avoid the contention for mutex. Thus, the next scheduler (#3, numbering does not follow the story line here), provides local unbounded intrusive queue per kThread and removes the global ReadyQueue. The scheduler assign the uThreads to kThreads in a round-robin manner. Each queue is protected with a std::mutex and std::condition_variable, and the following figure shows the design: and here are the results: Removing the bottleneck and getting rid of the central lock seems to provide better scalability, but can we do better?

- **Local RunQueue per kThread using lock-free non-intrusive Multiple-Producer-Single-Consumer Queue**: Since the only consumer for each local queue, is a single kThread, to reduce the synchronization overhead it is better to use a lock-free Multiple-Producer-Single-Consumer queue. The queue that is being used is a non-intrusive queue (you can find an implementation in the source code or `here`). With this queue, there is no contention on the consumer side and producers rarely block the consumer, and to push to the queue producers using an atomic exchange. Here is the result for using Scheduler #4:

- **Local RunQueue per kThread using lock-free intrusive Multiple-Producer-Single-Consumer Queue**: To avoid managing an extra state, the above queue is modified to be intrusive, and here are the results for using the intrusive queue:

For lower number of threads the global queue seems to do better, but as the number increases the intrusive lock-free MPSC Queue is a better Choice. The default scheduler for uThreads is Scheduler #2 (intrusive MPSCQ), which can be changed at compile time by definding **SCHEDULERNO** and set it to the appropriate scheduler.

You can add your own scheduler by looking at the source code under *src/runtime/schedulers*, and provide a scheduler number and a Scheduler class in its own header. Documentation for adding a new scheduler will be added as soon as the code base reaches a stable state.

Migration and Joinable uThreads

uThreads can be joinable, which means upon creation the creator uThread can block until they finish execution. There are three ways to execute a piece of code on another Cluster (These can be used to execute tasks asynchronously on current Cluster or a remote one):

- **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 state is saved in the stack and when migrated the state is resumed from the uThread's stack. The following code demonstrates a simple scenario of migrating 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);
.
.
.
}
```

- **Non-joinable uThread:** Create a non-joinable uThread on the remote Cluster and wait for it to finish execution. While waiting, the creator uThread continues execution and does not care about when the new uThread finishes the job.

```
Cluster *cluster1;

void run(){
    //code to run on cluster1
}
void func(){
    // some code
    uThread *ut2 = uThread::create(false); //create a non-joinable thread
    ut2->start(cluster1, run);
    //continue executing
 // some more code
}


int main(){

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

- **Joinable uThread:** 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 when the new uThread finishes its job. Currently, only the creator can wait on the new uThread, waiting on the uThread from other uThreads leads to undefined behaviour. This will be fixed in the near future.

```
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.

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`.

Acknowledgement

This work made possible through invaluable helps and advices I received from `Martin Karsten`.