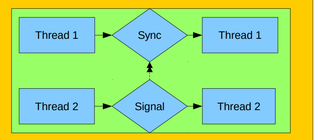
Implementing Pthreads for the XMOS Xcore

Synchronisation Primitives

At the most simplistic level, all synchronisation primitives can be abstracted down to two actions:

- A synchronising action, where a thread waits until it receives a signal
- A signal action, where a thread signals for thread(s) waiting on a synchronisation primitive to continue

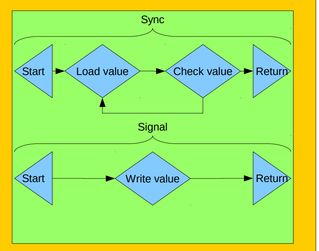


The Xcore architecture features a number of hardware locks, that provide a fast and simple method of synchronising threads, with both the synchronising and signalling action being a single instruction. However, they are limited in number and they do not support timed locking unlike more advanced pthreads synchronisation primitives.

The most basic pthreads synchronisation variable is the spinlock, where a thread performing the synchronising action constantly reloads a value from memory until another thread signals by adjusting the value to allow the synchronising thread to continue.

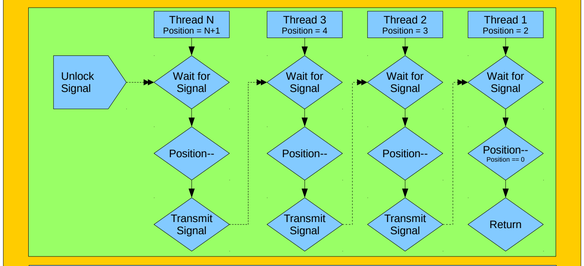
They have small memory footprints and allow any thread to lock at any point, but they are wasteful of CPU time and energy – the latter being especially important for embedded systems.

In this pthreads implementation, spinlocks are used to ensure that threads are added to more complex synchronisation variables in sequence, avoiding race conditions.

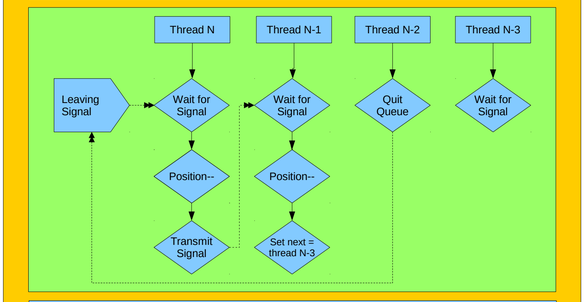


A more complex synchronisation variable is the mutex, where threads can lock the mutex before unlocking it for the next thread. Whereas a spinlock keeps a thread alive for the duration of the synchronising action, a mutex allows a thread to be paused and removed from the list of runnable threads until a signal is received.

As mutexes only wake a single thread at a time, it is necessary to maintain an ordered collection of threads waiting on each mutex. An early implementation featured a standard queue in memory, but this became unreliable to work with when the various situations where threads can exit a mutex queue were implemented. Instead, the design was changed to a system that keeps all the values needed in registers and uses the Xcore's channel system to pass an unlock signal along to the next thread to be activated.



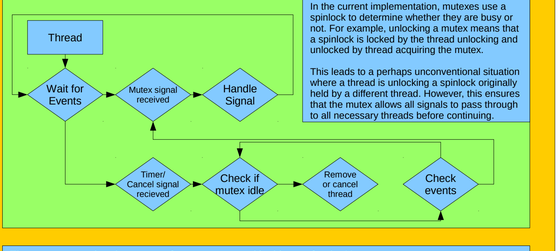
When threads exit the queue, they transmit their old position and the thread after them in the queue across the queue, which allows the thread behind them to repair the queue without having to store the state of the queue in memory outside of each thread's registers.



Threads need to be able to leave a mutex in two cases:

- When they are asynchronously cancelled (
- If they are performing a timed lock rather than a standard one

However, both these actions can occur when a mutex is having a signal passed through it – and a thread suddenly leaving the queue when a signal is being passed through the mutex will leave it in an usuable state. To avoid this, both these exit conditions need to check whether there is a signal being sent and handle that request before they can remove the thread from the queue.



In the current implementation, mutexes use a spinlock to determine whether they are busy or not. For example, unlocking a mutex means that a spinlock is locked by the thread unlocking and unlocked by thread acquiring the mutex.

This leads to a perhaps unconventional situation where a thread is unlocking a spinlock originally held by a different thread. However, this ensures that the mutex allows all signals to pass through to all necessary threads before continuing.

There are two main advantages of mutexes created in this way for the Xcore:

- As the work is more evenly divided over several threads, locking and unlocking the mutex is much quicker from the perspective of the calling thread
- Far fewer memory reads are required as almost all values can be kept in thread registers, and a smaller memory footprint.

The XMOS XCore

The XMOS Xcore is a range of embedded processor intended for fast reacting parallel programs.

They feature a number of features that help them carry out much of the functionality normally delegated to operating systems in hardware, resulting in much faster responses to input compared to competing processors.

Additionally, the structure of the core allows operations to have predicatable run times even when mutliple cores are active.

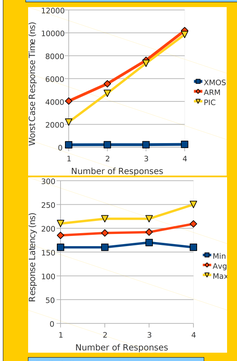


Fig. X – An Xcore processor tested against comparable ARM and PIC processors for response to an input
Fig. Y – The reponse times of an Xcore processor to input

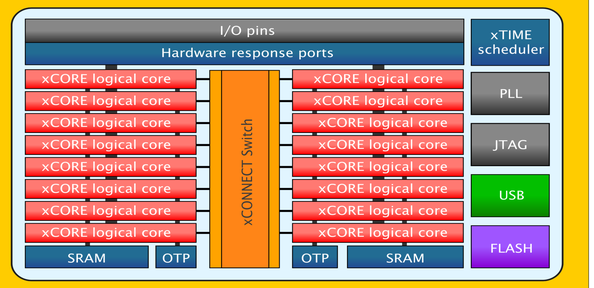
XMOS provides a compiler, XCC, that allows C and C++ code to be compiled for the Xcore.

They also have developed their own variant of C, called XC, which allows much easier access to the unique functionality of the Xcore.

Whilst similar to C in syntax, XC lends itself to a different style of programming to that which pthreads is designed for.

Most notably, XC has strict rules about pointers and pointer safety to avoid race cases and non-thread safe variables.

XC also does not support floating point operations, as the Xcore has no native floating point hardware, instead relying on the XCC C compiler to insert the relevant code to manage floating point operations.



A defining feature of the Xcore and one that helps enable it's rapid response times is the channel communication system used to pass data between threads.

Channel communication is very fast, with threads on the same processor being able to pass messages with delays of a few nanoseconds. Even between cores, the delay typical does not extend beyond a few milliseconds.

Threads waiting to read data in from channels are automatically removed from the set of runnable threads by the hardware, which allows them to pause and continue once they have been given an input.

Channels can also be use to trigger events and interrupts on a thread, allowing threads to respond to events even when in the middle of other tasks.

The data ports on an Xcore can be used in similar ways to channels to trigger events and cause threads to wait, though they are naturally slightly slower in response time.

The Xcore also has hardware level timing resources, that allow threads to wait for certain durations or to trigger events after time has elapsed.

Pthreads

Pthreads is a parallel programming paradigm that has arisen as part of the POSIX operating systems standard.

Whilst originally native to POSIX, various ported versions of it now exist for other operating systems such as Windows.

Compared to other parallel programming paradigms such as OpenMP, it allows for more low-level manipulation of threads with the accompanying requirement of the programmer managing more of the threading themselves.

The official pthreads standard consists of 100 functions, 13 types and 24 defined flags, with a variety of non-portable functions also included in many distributions.

Much of the pthreads functionality concerns synchronisation primitives rather than the creation of threads themselves, leaving much of the backend implementation open to interpretation.

Thread creation and destruction

Thread Memory Allocation

On larger systems, creating a thread often allocates a new stack from the heap for each thread. On embedded systems, this leads to major problems as memory fragmentation builds up, especially with how large thread stacks can be.

Instead on the Xcore, we statically reserve a large chunk of memory at during compile time and each created thread gets given a chunk of this space. Running out of space simply causes the program to exit, as there is no reliable system for allocating more stack space.

As larger systems allocate memory for threads dynamically, they can keep relevant thread data alive for the duration of an application's runtime. However, a statically allocated system will eventually need to re-use space if threads are created and destroyed repeatedly.

Thread Id

There are actually three different values that can be considered thread ids in the current implementation.

The first is the hardware provided thread id, which is important as it is the only way a thread can find its way back to its own data structure, by doing a lookup through all the thread headers matching the recorded thread ids against the one provided by the hardware.

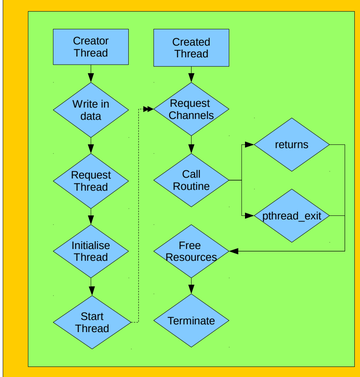
The second is the index of a thread in the static data structure. This is useful because it is the quickest way to find relevant data for a thread, but it cannot be relied on as a reference to threads outside of the one who owns it as threads are created over old disused ones.

The third one is the composite id of a thread, and is what is used where thread ids are needed by the pthreads specification, is a work around to the problem of reusing thread space.

To make the composite id, you take the index of a thread in the data structure and add the number of times that a thread has used that slot already multiplied by the maximum number of threads the implementation is configured to hold.

This means that you can quickly work backwards to get the actual thread data by taking a modulo, whilst also being able to check whether the thread has been overwritten

DIAGRAM



Thread Initialisation

Whilst much of the data a thread needs can be written in by the creating thread, there are certain tasks that the new thread must perform before before being able to continue

An example of this are the two channels that a thread needs to acquire.

Thread Termination

Threads terminate on their in own in two situations – when their main routine finishes or when they call pthreads_exit.

In the current implementation, both of these lead to calling the same function, which ensures that cleanup is handled properly.

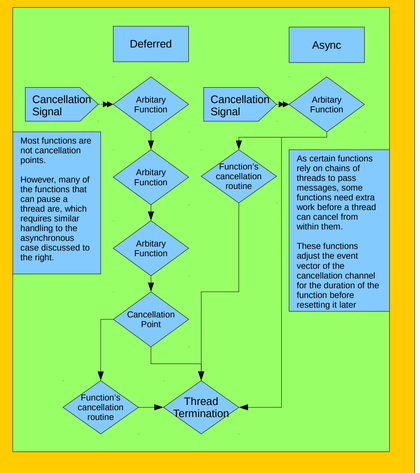
Thread Cancellation

Pthreads defines two cancellation types, asynchronous and deferred.

Asynchronous cancellation allows a thread to be cancelled by the implementation at any point, whereas deferred only checks at predefined or programmer-specified points.

Both cancellation types are implemented through each thread having a channel through which other threads can signal for cancellation.

There are occasional points where asynchronous cancellation is disabled briefly whilst reconfiguring the event vector and inserting the thread into a synchronisation primitive, to avoid situations where the thread being suddenly cancelled would cause synchronisation primitives to become unusable.



Developement Tools

Language

Whilst the XC language is normally the easiest way to manage the Xcore's unique functionality, standard C was used as the language for this project.

For the Xcore specific functionality, inline assembly was used, with the result that many functions contain substantial sections of assembly rather than C code.

xTime composer

XMOS provides xTime composer, an IDE specifically for working on the Xcore that helps manage and compile projects for XMOS products.

Xtime composer is also distributed alongside variations of the standard GCC toolchain for working with the Xcore.

One tool that has been very useful for debugging some of the more unusual bugs that have appear has been xobjdump, which decompiles the result of XCC's compilation.

Despite the generally very high quality of XMOS tools, there are very occasional features they are missing that cause problems; the lack of an *asm goto* statement in particular.

Implementational Difficulties

Compiler Support

Whilst the compiler for the Xcore supports C, C doesn't support many of the more unusual features of the Xcore.

One of the most obvious features of this is that assembly coding needed for many functions is, to the compiler, non-functional.

Various bugs have occurred where the compiler unexpectedly tries to optimise functions in ways that clash with inline assembly, and these can be difficult to find.

Because elements of the pthreads standard are incompatible with XC (for example, the need for void pointers in thread creation and joining), there are occasional places where event handling has been needed in C.

The compiler is not able to interpret the jumps that event handling does, which requires manually recalculating offsets, and awkward work arounds to avoid the compiler removing unreachable code.

Thread-safe debugging

In multithreaded applications, only one thread can print at a time.

This means the standard debugging technique of scattering printfs in critical points can and does actually change the code's functionality.

Testing the pthreads implementation

Testing the implementation

Much of the more complex code in this project relate to the Xcore's unique hardware, especially the channels.

Channels cannot be freed if they have outstanding data to be read from them, or if they have outstanding data to be transmitted. Channels also pause if they try to read when there is no data.

This combined makes working with channels fail very quickly if anything is out of order.

Each small collection of functionality (e.g. all the mutex functionality) has been tested individually, and designed to leave all shared resources in a usable state for further functions.

XMOS's development tools also feature debug capabilities that have been use to ensure that unintended behaviour is avoided.

Comparison to standard XC programs

One of the core questions for this project has been whether there are situations where a pthreads implementation is more useful than simply rewriting a code base into XC.

Because XC and C can be compiled together and C functions can be called from XC (and vice versa), a direct port of pthreads to XC can be as simple as changing the main function to an XC *par* statement rather than a series of pthread_create calls.

However, programs that rely heavily on the more advanced functionality of pthreads have more difficulty being ported into XC. The lack of native advanced synchronisation primitives being the prime example of what XC lacks.

Pthreads also lends itself to a different programming style than XC, which is probably due to its heritage as a parallel programming paradigm for more advanced systems.

Pthreads often wants to create threads in response to events where the thread completes a task before terminating. XC however, normally wants to take advantage of the Xcore's message passing systems and have threads with long life times and have data passed into them.

Performance wise, there is a slight advantage to XC in most situations, mostly due to the compiler being more able to optimise thread creation compared to the pthreads implementation. It is likely that there are a few changes that can be made to the pthreads implementation to improve this, but it is unlikely that the gap can be closed.

Even if the pthreads implementation is not directly useful, which seems unlikely considering the time required to port larger pthreads programs to XC, many of the functions in the collection have the potential to be used in hybrid programs that need certain functionalities.

Non-implemented functionality

Scheduling

- A core assumption of pthreads is the existence of a POSIX operating system on the processor, which is not the case with the Xcore.
- POSIX systems allow priorities to be set for different processes and threads, allowing some to proceed faster than others.
-
- The Xcore has no direct way to mimic this functionality, and a selling point of the Xcore is the guaranteed even spread of processor time between threads.
- Pthreads assumes that threads can be scheduled in several different patterns such as FIFO and Round Robin.
- The Xcore's hardware level scheduling means that in order to manage this, you would have to design an entire operating system to manage pausing and unpausing threads.
- The Xcore is typically used over other embedded systems for its predictability and the channel system both of which would be made more difficult to use if the operating system were able to interfere with the running state of the program.
- Pthreads need for scheduling is historically tied to having more threads than logical cores, which this implementation avoids.

Dynamic thread memory allocation

- The pthreads standard describes a collection of functions for adjusting the stack size of a thread during thread creation.
- On a larger system with more memory, this is easily managable with malloc-style systems.
- Smaller embedded systems struggle with large memory allocations due to fragmentation, especially if intended to run for long periods of time.
- The pthreads standard does not clearly give an expected lifespan for the extra data needed by each thread, which feeds into potential fragmentation issues and memory leaks.
- This pthreads implementation instead uses statically allocated memory to provide space for each thread.
- This avoids fragmentation and memory leaking in exchange for having a hard limit on how many threads can be concurrently active.
- It also means that some of the data about threads have a limited lifespan after the thread has terminated. Data destroyed in this way is indicated to the program, but it runs counter to the spirit of the standard.

Additional functionality

Batch thread creation

The Xcore has a number of hardware synchronisers available that allow for a single parent thread to coordinate several children threads.

It is difficult to directly map this onto pthreads functionality as synchronised threads need additional attention from their parent thread to function properly, which would require programmers to deviate from a normal pthreads program to manage.

Additionally, it is programmatically awkward to remember and re-use a single synchroniser between different calls to pthread_create()

Instead, an additional function has been added that allows for the creation of several threads at once by a master thread bound to a single synchroniser

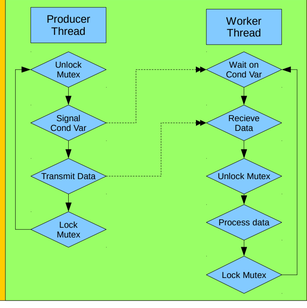
Other functions were added to allow the master thread to synchronise and join with the threads created in this way.

Condition Variable return values

In standard pthreads, condition variables only return a straightforward integer error code on calls to signal to waiting threads

In this pthreads implementation there is an additional variant function for signalling which return the channel of the thread woken by the condition variable.

This allows a condition variable/mutex combination to be used to pass data from a producer thread to worker threads easily.



Key

References