Hanging by a Thread

Lightweight Threads and Asynchronous I/O for the Modern Era

Hamish Morrison (hm53@st-andrews.ac.uk)
Supervisor: Dr. Graham Kirby

April 10, 2015

University of
St Andrews

Abstract

Threads provide a simple and familiar sequential model for programmers. However, when it comes to highly concurrent applications like web servers, using a thread for every connection can prove prohibitive due to overhead of thread switching and scheduling. In these areas event polling solution are often used instead — with a single thread dealing with hundreds or thousands of connections. This model can prove difficult to program however, and we lose the locality of threads. I develop and benchmark a lightweight user-level threading library, which attempts to provide the benefits of the sequential threaded programming model, with performance close to that of an event-based system.

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 10,042 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

Chapter 1

Introduction

An interesting problem in computer science is that of developing highly concurrent I/O-bound applications like servers. It is necessary to carefully select a granularity of concurrency and a programming model to balance scalability and performance with the ease of development.

Traditionally I/O bound server applications (e.g. the Apache web server's worker and prefork models[9]) used a thread-per-request or process-per-request model. The server listens continuously on a socket and when a new connection arrives, it forks a new process or creates a new thread to deal with that connection. All I/O operations performed are synchronous, with the operating system blocking the calling thread until the request completes or an error occurs.

The other option on UNIX-like systems of the 1990s was non-blocking I/O with event-based readiness notifications. Sockets are placed in non-blocking mode and function calls that would normally block, like `send` (for sending data on a socket) and `recv` (for receiving data from a socket), instead return an error code indicating that the operation could not be performed immediately. The application then uses a event notification interface, such as the `select` or `poll` system calls, to wait until the socket becomes readable or writable. These interfaces have some scalability issues however. Due to their stateless nature, every call to `poll` or `select` requires the entire set of file descriptors being monitored to be passed in and checked, making the calls O(N) in the number of descriptors. `select` is particularly egregious, as it uses an array indexed by file descriptor number, requiring a large fixed-size array that has to be copied between user and kernel space on every call, and also limiting the highest-numbered file descriptor that may be waited on.

The limitations of these interfaces were addressed by Banga, Mogul and Druschel in 1999[3], who proposed a new stateful interface, where the application declared interest in specific file descriptors and would make a call to retrieve any events — or wait for one — at a later point. The kernel would maintain a queue of events that had occurred on the file descriptors, and deliver these to the application when requested. The complexity of the wait call was therefore O(N) in the number of events, rather than the number of descriptors. In the network stack, each socket would have a list of the threads interested in it and dispatch events to each of them. The authors implemented the interface for

Digital UNIX V4.0D, and showed significant scalability improvements over the existing APIs. The benchmark measured request throughput with increasing numbers of inactive connections (i.e. connections that were being monitored for events). The new interface maintained a consistent request throughput, while `select` performance got worse with more inactive connections to monitor.

The results inspired the creation of similar APIs for other operating systems. In 2000, Jonathan Lemon developed an interface called `kqueue` for FreeBSD[15]. Lemon extended the idea to allow monitoring more than just file descriptor events; `kqueue` could also wait on file system events, AIO completions, and process start/termination events. The event queue was no longer coupled to individual threads; it was now created with a separate system call and referred to using a file descriptor of its own, allowing more than one queue per thread, and allowing the queue itself to be monitored as part of another `kqueue`. The new interface also offered a selection between edge-triggered (where an event is returned once — e.g. when a file descriptor first becomes readable) and level-triggered (where an event will continue to be raised while its condition holds true — e.g. while a file descriptor remains readable) event notification. While the Linux developers viewed `kqueue` as an over-engineered solution, they developed their own analogue called `epoll`, introduced in Linux kernel 2.5.44 — providing file descriptor monitoring only[8].

It is interesting to note that Microsoft preempted all these interfaces by quite a few years with their 'I/O completion ports', introduced in Windows NT 3.5[22]. I/O completion ports deliver notifications of I/O *completion*, with the actual I/O operation being performed by the kernel. This is in contrast to `epoll` and `kqueue`, which simply notify the user application that the file descriptor is now *ready* — that the application may now perform the I/O operation without blocking. Naturally this requires more complicated buffer management than `epoll` and `kqueue`, as the Windows kernel must take control of the buffer being used for I/O until the operation completes.

There was interest in these new event notification interfaces, particularly due to the C10K problem (the problem of developing a server capable of serving ten-thousand concurrent requests, documented very thoroughly by Dan Kegel[13]). The new interfaces made their way into web servers, such as Igor Sysoev's `nginx` in 2002 and Jan Kneschke's `lighttpd` in 2003. Benchmarks showed improvement over the old `select` and `poll` interfaces, and considerable improvement over the forking and threaded models of web servers like Apache[14].

These servers were programmed as state machines. Each connection had a state associated with it, and this state in conjunction with I/O readiness notifications would determine what actions would be taken next. Another abstraction for implementing such servers is through the use of callbacks. An asynchronous I/O request is issued along with a function to be called when the operation completes. The called function examines some piece of state and determines what to do next. While the performance of such techniques is good, the control flow is convoluted.

There have been numerous attempts to marry the performance of these event-based interfaces — `epoll`, `kqueue` and completion ports — with the convenience and familiarity of multithreading. One approach is to schedule threads and perform context switches purely in userspace. This way it may be possible

to avoid some of the overheads of kernel threading — the cost of mode switching, scheduling and resource allocation for stacks, thread-local storage (TLS) and kernel data structures. Placing thread scheduling within the application's control may also allow the application to make better scheduling choices, as the kernel is generally unaware of the intentions of the user application.

The goal of this project is to produce a user-level threading library, broadly aimed at highly concurrent I/O bound applications like servers. The library will be written in C, allowing for direct interfacing with the highest-performing system primitives — like futexes (for developing fast synchronisation primitives) and `epoll` — and allowing for careful optimisation of the critical paths like context switching. There is also a relative paucity of user-level threading C libraries, particularly ones that are also multithreaded — perhaps because they are hard to do right, or perhaps because there is more interest in developing lightweight threading as part of a runtime for a higher level language.

The next section explores previous work on threading libraries — user-level and otherwise.

Chapter 2

Context Survey

2.1 Operating system threading models

A thread generally consists of a thread execution state — the current set of registers and stack state — and some amount of kernel state — a thread ID, a signal mask (the set of signals the thread may receive), a scheduling priority and so on. An interesting question for operating system developers is what portion of thread scheduling to manage in kernel space, and what portion to entrust to user level. At one extreme of the spectrum, thread management can be done purely in a user space library, creating thread contexts and switching between all within the context of one kernel thread. At the other end of the spectrum, all management of threads is done through the kernel, and the kernel keeps track of all threads present in the system.

2.1.1 1:1 user/kernel thread model

This is the model used in most modern operating systems — Linux, Windows, Solaris and the BSDs. One user-level thread is mapped onto one kernel-level thread. Under this model, thread scheduling is preemptive and performed by the kernel. A timer interrupt causes a trap into the kernel, and the kernel makes a decision about whether to preempt the currently running user thread and replace it with another. When threads perform blocking operations, for example, reading from a socket, they are placed on a wait queue for that socket in the kernel and the kernel reschedules to some other thread. Eventually network stack code, on receiving some data for that socket, will wake the waiting threads, enqueueing them for scheduling.

This model easily takes advantage of multicore and multiprocessor architectures, as the kernel has a complete view of the CPUs available for it to schedule on. Modern operating systems have highly optimised scheduling algorithms — for example, Ingo Molnár's O(1) scheduling for Linux[1].

2.1.2 N:1 user-level thread model

This model is used by early threading implementations (e.g. FreeBSD 4's `pthreads` implementation, `GNU Pth`) and is implemented entirely in user space. All threads are mapped onto a single kernel-level entity. Thread scheduling might be preemptive or cooperative. Thread switching is very fast as it is done entirely in user space and doesn't involve a system call. Where possible, blocking system calls are replaced with non-blocking equivalents (perhaps using one of the event notification mechanisms previously discussed). Since every thread maps to the same kernel-level entity, however, this model cannot take advantage of multiprocessor systems. Additionally, operations that cannot be replaced with non-blocking equivalents will end up blocking the entire process. Since the kernel is completely unaware of the existence of multiple threads, it cannot schedule another in place of the blocked one, which limits the effectiveness of this model considerably.

Such implementations, which included FreeBSD's `libpthread`, and early versions of the Sun JVM, fell out of favour and have been superseded by 1:1 threading.

2.1.3 M:N hybrid thread model

Multiple user level threads are mapped onto a smaller number of kernel threads. This scheme tries to retain the advantages of the above two models without the disadvantages. Previous attempts (e.g. NetBSD's scheduler activations, and FreeBSD's kernel scheduled entities, both from the early 90s) have attempted to put scheduling control into the user application's hands through use of 'upcalls'[2]. Under this model, whenever some scheduling event occurs — e.g. a thread blocks or becomes runnable — the kernel makes a call into some designated user code (a 'scheduler activation' in NetBSD parlance) which then makes some scheduling decision and performs a context-switch itself. The user application makes system calls to keep the kernel updated with the number of user-level threads currently in existence.

Like the pure user-level threading approach, this model also fell out of favour on the major operating systems. NetBSD's implementation had difficulty scaling on multicore systems, and was replaced with a 1:1 threading model in NetBSD 5.0[17]. FreeBSD switched to a 1:1 threading model for FreeBSD 7.0. Julian Elischer, a developer of FreeBSD's M:N threading library, said that "many of the promissed [sic] advantages [of M:N threading] turn out to be phantoms due to the complexities of actually implementing it."[7]

There has been some recent interest in a scheduler activation-like approach for threading with the rise of 'library' operating systems. Barrelfish — a research OS aimed at multicore and many-core machines — uses a form of scheduler activations[4]. The kernel calls upon a user-level 'dispatcher' to schedule its own threads or tasks. There is a library that provides a familiar pthread-style library on top of this.

Google have also proposed a set of system calls for providing user-level scheduling, while maintaining the individuality of threads from the perspective of the kernel[19]. The core of the idea is the `switchto_switch(pid_t thread)`

system call, which atomically switches to the specified thread, leaving the current one unscheduled. From the kernel point of view, the first thread yields its timeslice to the second. With this system call, the individuality of threads in the kernel is maintained, allowing TLS, thread IDs and debuggers to continue to work as expected, while the cost of scheduling in the kernel is avoided. Blocking system calls are handled with a 'delegate' thread, which is switched to whenever current thread blocks. It is not clear exactly how user code is notified of threads becoming unblocked, as documentation is limited and Google have not made any of the code available.

2.2 Threading in programming language runtimes

Some programming language runtimes implement a M:N or N:1 thread scheduling model on top of the operating system's native thread functionality. They typically use an event notification mechanism as discussed above to avoid blocking kernel threads when user-level threads need to perform I/O.

2.2.1 Go's 'goroutines'

Go performs cooperative scheduling of its own user-level threads (goroutines) onto a few kernel threads[6]. When a goroutine performs a system call, its kernel thread will become blocked, so a new kernel thread is spawned to serve any goroutines that are still runnable. In the worst case, as many kernel threads as goroutines could be required. The runtime is generally quite successful in avoiding this, by using non-blocking operations and avoiding additional system calls wherever possible.

However, even system calls that are traditionally thought of as non-blocking can cause more native threads to be spawned if they are called frequently — the runtime does not distinguish between blocking and non-blocking system calls when it decides whether to spawn a new thread. One such example is the `ioctl` issued after `accept`ing a new connection to make the socket non-blocking. In intensive applications, users of Go found the runtime creating many hundreds of native threads as many were engaged in `ioctl` calls[21]. This was solved using `accept4` to set the accepted socket to non-blocking straight away, and to mark certain system calls as 'short', so their execution wouldn't cause more threads to be spawned.

Outside of system calls however, the number of kernel threads allowed to run at once is limited by default to 1 — although this can be configured with the GOMAXPROCS environment variable. Since Go 1.2, the runtime implements a simple form of preemptive scheduling. Upon function entry a call is made to the scheduler to check if the goroutine should be preempted. If a long-running loop does not make any function calls however, it will block all other goroutines. Naturally this kind of preemption is only possible with cooperation from the entire toolchain — the compiler needs to insert calls at function entry to check for preemption. This technique is therefore impossible to achieve at the library level; and even with a compiler extension, any calls into foreign libraries will be

non-preemptible.

Go also uses segmented stacks to reduce the memory overhead of each goroutine. Stacks consist of multiple linked segments. On function entry, the runtime checks if there is enough space for the stack frame in the current stack segment. If not, a new segment is allocated and linked into the stack. This means that goroutines only allocate stack space as it is needed. For C code, operating systems and C libraries will often allocate a large stack area upfront, as it cannot predict how much will be needed. It is possible, with the use of guard pages, to implement an expanding stack for C programs. The kernel responds to page faults on the guard pages by expanding the stack. However, such stack expansion may be limited due to address space collisions, a limitation not present in the Go runtime's linked-list of stack segments approach.

2.2.2 Erlang's processes

Erlang, like Go, has lightweight green threads, which it calls processes[16]. The virtual machine spawns a native thread for each real processor, and each of these native threads runs a scheduler. Each scheduler has its own run queue from which it takes and executes process jobs. Like in Go, non-blocking operations are used to avoid blocking on a system call. File I/O, and other operations that cannot be made non-blocking are farmed out to a threadpool.

Preemption is accomplished in the scheduler by way of allowing each process a 'reduction budget'. A process spends its reductions by calling functions, sending and receiving messages and so on. The scheduler switches out the process after it either spends its budget or needs to wait (e.g. for a message). Since Erlang is an interpreted language, such a preemption check can be trivially implemented as part of the interpretation process, coming basically for free, while in a compiled language like Go this is more difficult, as inserting preemption checks at arbitrary points in the code could have a significant performance impact, growing loop bodies and trashing the cache.

Chapter 3

Design and Implementation

3.1 Goals

The primary objective of the project is to implement user-level threading in the form of a C library. The library should provide lightweight threads with context switching performed in userspace. The library should provide lightweight-thread compatible replacements for blocking functions (e.g. `send`, `recv`, `sleep` etc.) that will block only the current lightweight thread and reschedule to another one behind the scenes, without blocking the kernel thread.

The library should be multithreaded, allowing it to take advantage of multiple logical processors for scheduling lightweight threads on. Additionally, while the primary development environment and target will be Linux and x86-64, the library should be portable to other operating systems and architectures and not be excessively tied to specific operating system features.

3.2 Design

I envisioned the core of the library being the scheduler, a per-CPU entity that would be responsible scheduling the lightweight threads, maintaining run queues and keeping track of blocked lightweight threads. Users of the library would interact with the scheduler through the asynchronous I/O routines, which have a similar interface to the regular synchronous I/O system calls. The user should never interact with the scheduler directly, and its initialisation should be invisible to the user.

At the core of each scheduler would be an event loop. The event loop would query an event notification mechanism for events and schedule any lightweight threads as appropriate — i.e. enqueue them on them on the scheduler's run queue. The event loop would then take the first lightweight thread and execute it. With preemptive scheduling enabled, the currently running lightweight thread should be preempted at some interval. The scheduler should then save the thread's state and enqueue it to the back of the run queue, and execute the next one from the head of the queue.

3.3 Implementation

3.3.1 Context switching

For the first steps of implementation I started with the very basics: context switching. The most obvious functions providing this functionality are the `setjmp` and `longjmp` functions, which are part of the C standard[11]. Unfortunately however they lack the ability to easily create new contexts (on a new stack and with a new initial stack frame) to jump to. The POSIX standard provides a family of functions for switching user contexts called `ucontext`. Despite this interface now being deprecated (due to it using a language feature marked obsolescent in C99[18]) it is widely supported among UNIX-like operating systems.

The `ucontext_t` structure provides storage for the architecture-specific thread state, along with the signal mask and stack information. The `makecontext` function can be used to create a new context for a lightweight thread, given a region of memory for the stack and an initial function to execute. The `swapcontext` function performs the context switch — writing the current context into the first `ucontext_t` parameter, and switching to the second `ucontext_t` parameter. The function effectively does not 'return' until the first context is switched back to. These routines provided the basic building blocks for implementing thread switching.

From reading of others' experiences with these functions, I was aware of major drawback: they required a system call on every context switch in order to store and restore the signal mask (which was part of each `ucontext_t`). Therefore I wrapped up the `ucontext` code behind a small abstraction of my own, so I could swap it out easily at a later point in time if I found the performance poor. This was a good idea, as later on I found myself needing far more control than the `ucontext` routines allowed for.

To verify the basic context switching worked as expected, I created a simple API for creating threads and switching between them — essentially a cooperative threading API. This initial implementation used `mmap` to allocate memory for the 16KB thread stacks — as it seemed more appropriate for the large, page-aligned allocations than `malloc` — however this was also noted as another area for potential optimisation later on down the line.

3.3.2 Event loops and asynchronous I/O

The next step was to replace the blocking I/O routines with ones that would only block the current lightweight thread. The routines were designed to be used with sockets that are in non-blocking mode. The routines operate by executing the usual system call — e.g. `recv` or `send`. If the call succeeds then the lightweight thread can just continue with execution as normal. If the system call returns either `EAGAIN` or `EWOULDBLOCK` however, this signals that the I/O operation would have blocked. The lightweight thread registers itself with the event loop to be awoken when the right event occurs (e.g. when the file descriptor becomes readable or writable) and then switches to the event loop lightweight thread.

Registration involves creating a small 'waiter structure' on the stack, storing a reference to the lightweight thread to be resumed, and the file descriptor and events that are being waited on. The file descriptor is then added along with the waiter structure to the underlying event notification mechanism — e.g. the epoll or kqueue descriptor. The thread then context switches to the event loop thread. When the event occurs and the lightweight thread is awoken by the event loop, it reattempts the I/O operation. This can potentially race with other threads reading and writing the same file descriptor, and the I/O operation can fail again with `EAGAIN` or `EWOULDBLOCK`. Therefore it is necessary to perform this in a loop. See listing 1 for the implementation of `async_write`.

```
ssize_t async_write(int fd, void* buffer, size_t length) {
    while (true) {
        ssize_t num_written = write(fd, buffer, length);
        if (num_written >= 0) {
            return num_written;
        }

        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            event_loop_wait(fd, WRITABLE);
        } else {
            return -1;
        }
    }
}
```

Listing 1: Implementation of `async_write`

The event loop thread consists of a simple loop which polls on the event notification mechanism, and then schedules any lightweight threads that are now able to continue with I/O operations. The event loop maintains a linked list of runnable lightweight threads, and these are scheduled in FIFO order. If this list is non-empty, the event loop polls the event notification mechanism in a non-blocking manner — picking up any pending events but without waiting if there aren't any. It then enqueues any unblocked threads to the back of the list, and switches to the thread at the head of the list. If the runnable list is empty, the event loop will block indefinitely on the event notification mechanism until some thread becomes ready.

To verify that these basic I/O routines worked correctly, I created a simple TCP echo server which runs a server and a number of clients which send a series of messages to the server and read them back.

3.3.3 Timers and sleeping

Another basic blocking operation that needed to become lightweight thread aware was timed sleeps, which are useful for implementing any timed operations. One can of course use signal-based timers, which will continue to work normally in the presence of lightweight threading, and file descriptor-based timers (i.e. Linux's `timerfd_create` API), which can be integrated easily with the event

notification mechanism used for the asynchronous I/O routines. But also of interest is a lighter weight mechanism for sleeping a lightweight thread. To accomplish this I added timers to the event loop API.

The event loop was extended to maintain a list of timer structures in a binary heap structure, indexed by their absolute expiration time. The timer structures record their expiration time, and the lightweight thread that should be woken up when they expire. This allows insertion and removal of timers in O(log n) time (in the number of timers), and O(1) time finding the next timer that will expire. The event loop then uses the timer at the top of the heap to determine what timeout it should use when waiting on the event notification mechanism. Every time round the loop, the timers at the top of the heap are checked and expired if necessary, rescheduling any lightweight threads.

There are better performing timer algorithms, such as the O(1) timer wheels described by Varghese and Lauck[20], where timers are stored in a hierarchical set of bucket arrays indexed by millisecond, second, minute and so on. However I did not pursue this implementation, as it was not particularly important to the benchmarks I was planning to run.

3.3.4 Preemptive scheduling

I was also interested in providing optional preemptive scheduling of lightweight threads. Mechanisms used by managed languages, such as preempting lightweight threads upon function entry or some common operation like memory allocation, are not possible for a C library. The main problem is to interrupt execution of some code over which we have no control. One obvious mechanism to accomplish this is signals — available on all UNIX-like operating systems.

Signals require cooperation from the kernel to operate. The kernel maintains a queue of signals for each process — which can be delivered to any of the process's threads — and a queue of signals for each individual thread. Eventually a user thread has to give up control to the kernel, either to make a system call, or as a result of some interrupt (including timer interrupts for kernel scheduling). When the kernel prepares to return to user mode, it checks the thread for any pending signals. If there are any, the registered signal handler is executed on the stack of the interrupted thread. After execution of the signal handler, the original thread state is restored and control is returned to the instruction previously interrupted.

Therefore, to implement preemption on UNIX-like systems I chose to use signals. There are a number of timer functions that can be used to send a signal on expiry — `setitimer`, `alarm` and the newer POSIX timer functions (`timer_create`, etc). From the signal handler, it is then possible to 'jump out', context switching to another lightweight thread. The in-progress execution of the signal handler is stored in the context of the old, interrupted thread. When this context is resumed (i.e. when the original lightweight thread is switched back to) the signal handler will complete execution and resume execution of the interrupted thread.

Documentation is quick to point out the dangers of calling non-signal safe functions in signal handler context, which is essentially what is done here, as the lightweight thread switched to from the signal handler can execute anything.

The danger of deadlock is avoided here however, because if a lightweight thread tries to take a lock held by some other lightweight thread, the timer signal will eventually be sent, and the lightweight thread holding the lock will eventually be rescheduled and complete the critical section.

This situation does raise some performance questions however. Consider for example, lightweight thread 1 holds some futex-based lock and is interrupted. LW thread 2 is scheduled in its place, and tries to acquire the lock. Finding it contended, LW thread 2 makes a `FUTEX_WAIT` system call and is enqueued on the futex's wait queue. The underlying kernel thread is now blocked on the futex, despite there being a runnable lightweight thread. Eventually the wait will be interrupted by the timer signal, interrupting LW thread 2 and allowing LW thread 1 to run once more.

As we can see, implementing preemptive scheduling purely as a library — while possible — may have some performance issues, particularly when calling into foreign code which is unaware of the lightweight threading. To avoid this issue, lightweight thread aware locks and synchronisation mechanisms would be necessary, and it would be necessary to modify all user-level code, from the C library up, to make use of them. The preemption feature thus remains an optional part of the library, disabled by default.

While I did not have time to implement a Windows port, I wanted to check that it is indeed possible to implement this on Windows. To accomplish user-level preemptive scheduling on Windows, a slightly different technique is necessary, as Windows does not provide signals. Using a second interrupt thread, it is possible to suspend the main thread. From there, using the Windows debugging API, the context of the suspended thread can be captured and saved, and a new context can be set.

3.3.5 Multithreaded scheduling

Another goal of the library is to take advantage of multiple logical processors. For server-type programs, where the amount of data shared between threads is minimal, this is easy to achieve using a multiprocess architecture. Multiple processes can listen on the same server socket and each deal with their own set of clients using lightweight threading. However this solution is insufficient for applications that require a shared memory architecture; for this it is necessary to add multithreading capabilities to the library itself.

To enable this, I extended the threading library to start a new pthread every time a new lightweight thread was spawned, up to the limit of the number of logical processors on the system. Each of these native threads has an associated 'local scheduler' structure, which is responsible for scheduling lightweight threads to run on it. Each local scheduler has its own run queue of lightweight threads, and an event loop thread to run when its run queue is empty. A process wide 'global scheduler' remains. It manages the global event notification mechanism, the heap of timers, and keeps track of how many local schedulers have been initialised.

Each local scheduler polls the event notification mechanism, and schedules now-runnable lightweight threads on its own run queue. It locks the global scheduler and checks its timer heap for any expired timers, enqueueing cor-

responding lightweight threads on its run queue if necessary. If there are no runnable lightweight threads then the local scheduler performs a blocking wait on the event notification mechanism. Clearly this is suboptimal if there are lightweight threads waiting on the run queues of other schedulers. Some form of work stealing could be implemented[5], so local schedulers without work would remove lightweight threads from the run queue of another scheduler. Such a scheme would maintain work locality as far as possible, with threads only being redistributed when absolutely necessary — when a scheduler is out of work. The present queue data structure — a doubly-linked list — is not amenable to doing this in a lock-free fashion however. Because work-stealing is not necessary for the I/O driven benchmarks, I have not implemented it.

With a server-like workload, where individual threads execute for short time periods before blocking on I/O, the event notification mechanism effectively handles the distribution of threads to schedulers, as it will return events to any scheduler thread currently polling. This could have consequences for cache locality, as lightweight threads may not be rescheduled to the same logical processor originally running them. This may be improved with some cooperation from the kernel — for example, having `epoll` prefer to return an event to the same thread that originally registered the file descriptor. At the same time, for a very high throughput server this may have little benefit due to a large number of connections polluting the cache in the meantime.

3.3.6 Race conditions and locking

The implementation of multithreaded scheduling uncovered an interesting race condition that only presents with multiple threads. The problem is the non-atomic nature of suspending a lightweight thread to wait for some event. A number of things need to be accomplished when doing this:

1. Saving the lightweight thread's context to be restored later.
2. Registering the lightweight thread with some event notification or timer mechanism to be resumed later.
3. Rescheduling and context switching to a different lightweight thread.

It is only after step 3 that it is safe to resume the lightweight thread, as its stack is no longer in use. It is possible however after step 2 for some other scheduler thread to pick up the thread (which is still in the process of suspending) from an event notification, or from the timer heap. If the other scheduler were to switch to the thread at this point, it would execute with the thread itself on the same stack, corrupting its data. One obvious solution to this is to give each lightweight thread its own spinlock. Any scheduler suspending a thread or resuming a thread would then have to take this spinlock before doing so. On context switch, the switched-to thread would have to do the job of releasing the spinlock of the switched-from thread. This is the strategy used in the scheduler of most modern operating system kernels.

Given that our needs are simpler than that of a kernel thread scheduler — with its larger amount of state and scheduling complexity for each thread — we can do better than this, and avoid using locks altogether. Using some atomic

operations, the process of putting a lightweight thread to sleep can be split into four steps:

1. Preparing to sleep. This atomically sets a status flag indicating that the lightweight thread is `BLOCKING`.
2. Registering the lightweight thread with the event notification mechanism or timer heap.
3. Saving the lightweight thread's context.
4. Sleeping, if no race occurred. This atomically changes the status from `BLOCKING` to `BLOCKED`, only if it wasn't changed back to `RUNNING` in the meantime.

When another scheduler wants to wake up a sleeping thread, it uses an atomic fetch-and-set to set the thread's status to `RUNNING`. If the operation reveals that the status was `BLOCKED` before the fetch-and-set, then that means the thread in question has already completed step 4 and can be safely resumed. If the previous status was `BLOCKING`, then the waking scheduler does nothing; it leaves the status as `RUNNING` and does not reschedule the thread. The thread, which is in the process of going to sleep, will notice at step 4 that is has been resumed. At this point it reverts its actions, and resumes executing the thread. Thus we have race-free multithreaded scheduling without any locking.

This work necessitated replacing the context switching solution I had used so far — POSIX `ucontext` — with a solution written in assembly. For step 4 of the sleep procedure, it is necessary to perform the compare-and-set of the thread status and the context switch to the next thread without touching the stack, because at any point after the compare-and-set it is possible for the thread to be awoken. The assembly implementation — currently implemented for x86 and x86-64 (see listing 2) — accomplishes this. The assembly implementation also avoids any potential performance problems with `ucontext`; it saves as little state as necessary and does not make system calls for saving/restoring the signal mask.

3.3.7 Thread death

Handling thread death poses an interesting problem: who exactly should free the resources (i.e. stack and thread control block) of the thread which has just completed? The thread cannot free these itself, as it needs a stack on which to execute the code to free the stack. One solution is to make this the responsibility of the thread that is switched to after the current one exits. The switched-to thread could, upon exit from the context-switching function check for any dead threads that need to be cleaned up. This solution does introduce some overhead to the critical path — the check must be made on every context switch.

Observing that only one lightweight thread can terminate at a time on each scheduler, each local scheduler can be given its own utility stack (one page in the current implementation). This utility stack can be set up with a small temporary context to execute — i.e. a stack frame pointing to a function to

```
context_save:
    # Save all callee-saves into the context
    movq %rbx, 0(%rdi)
    movq %rbp, 8(%rdi)
    movq %rsp, 16(%rdi)
    movq %r12, 24(%rdi)
    movq %r13, 32(%rdi)
    movq %r14, 40(%rdi)
    movq %r15, 48(%rdi)

    movq 0(%rsp), %rax
    movq %rax, 56(%rdi)

    xorl %eax, %eax
    ret

context_cmpxchg_restore:
    # Compare and set the value at the address
    movl %edx, %eax
    lock cmpxchg %ecx, 0(%rsi)
    jz   .L1
    # If the value at the address was not what we expected, return
    ret
.L1:
    # Restore all the callee-saved registers
    movq 0(%rdi), %rbx
    movq 8(%rdi), %rbp
    movq 16(%rdi), %rsp
    movq 24(%rdi), %r12
    movq 32(%rdi), %r13
    movq 40(%rdi), %r14
    movq 48(%rdi), %r15

    # Place the PC in our return address
    movq 56(%rdi), %rax
    movq %rax, 0(%rsp)

    movl \$1, %eax
    ret
```

Listing 2: Atomic thread suspend-and-switch implementation

free the resources of the terminated thread — and then reschedule to the next thread.

I noticed that this technique could also be useful for other synchronisation needs, in particular it could also be used to solve the atomic thread suspension problem discussed in the previous section. The thread suspending itself would save its context, and make a context switch to a temporary context on the utility stack. This temporary context could then proceed to do the work

of marking the suspending thread as blocked, and registering it with the event notification mechanism or timer heap. This solves the race conditions experienced previously, as the suspending thread is now perfectly save to resume as soon as it has context switched to the utility context. This removes the need to use costly atomic operations during thread suspension. However it does introduce the need to one extra context switch per thread suspension (although this could be optimised somewhat — essentially all we are doing is changing the stack pointer and executing a function). Regardless, while I did not have time to implement this alternative thread-suspension mechanism, it would have been interesting to see if this offers any performance improvements.

3.3.8 Compiler optimisation issues and TLS

Thread-local storage raises a problem for a user-level threading library; naturally, since a lightweight thread can migrate between different native threads, any TLS support provided by the C library or operating system (for example, any of the `pthread_key_*` functions) will be unusable. Any 'blocking call' — which under the hood context switches to the event loop — can potentially cause migration to another native thread. With preemption enabled, migration can occur at literally any point in the user's code. Proper thread local storage support for lightweight threads would require a specialised set of lightweight threading-aware TLS functions, and every piece of code that needed TLS would need to make use of them. Since TLS was not a requirement of any of the benchmarks I was planning to run, I did not implement any such functions.

Any foreign libraries (including the C library) that make use of TLS will face potential problems. One of the primary uses of TLS in a UNIX-like system is to store the `errno` — a number set by library functions on error to indicate what error occurred. The lightweight thread library therefore stores and restores `errno` when switching between lightweight threads, in an effort to mitigate this problem.

Compiler optimisations can complicate the picture further, particularly if `errno` is implemented as using a compiler extension such as GCC's `__thread` storage specifier, or Clang and MSVC's `__declspec(thread)`. In such a case, the compiler may chose to keep a reference to the storage location for `errno` in a register or on the stack, even though it may become invalid. Under glibc, `errno` is defined as a macro that expands to the value pointed by address returned by the function `errno`. One may think this avoids compiler optimisation problems — under the C standard the compiler can't cache the address returned by `__errno_location`. However, the function is marked with the `const` attribute, which allows the compiler to violate this.

```
extern int __errno_location() __attribute__((const));

#define errno (*__errno_location())
```

Listing 3: Effective errno definition in glibc

The obvious (and hacky) solution I have used is to hide access to `errno` with another macro, with expands to a call to a non-const function. Clearly

this requires all users to make use of this new macro, which is problematic for any calls into foreign code, which is unaware of lightweight threading and the volatility of the address of `errno`. Thus use of TLS raises serious problems for the threading library.

3.3.9 Optimisation

When I started developing and running early versions of the benchmarks, I took the opportunity to use `gprof`, a Linux call graph profiler, to determine what the costliest functions were — both in terms of their inherent cost, and in how frequently they were called. A clear culprit was the `mmap` call for allocating thread stacks. Replacing it with a call to `malloc` — which does not (usually) require a system call, and does not require manipulating address space structures — improved the situation considerably. Further gains could be made here — for example, caching the thread stacks and data structures of exited threads for reuse — but the `malloc` call was not particularly costly.

To guard critical sections of scheduler code, I had used a flag set on the local scheduler to temporarily disable preemption (the preemption signal handler would notice the flag and return straight away). I had used a full memory barrier for the flag, which turned out to be expensive, given that the flag is set on every context switch and scheduler action. I realised that the full memory barrier is not necessary at all, as the preemption signal handler is always going to *interrupt* execution of the scheduler code, never run concurrently with it. Therefore all that is required is a compiler barrier (which has no cost in itself) to prevent the compiler from reordering operations on the preemption flag with other operations.

After these optimisations no parts of the threading library were jumping out as particularly costly; most of the CPU time was now spent in the HTTP parser or polling for events in the event loop. I also noticed that `sched_init` — a one-time initialisation function which sets up the global scheduler — was being called very frequently. The function returns immediately if the scheduler is already initialised, but I had expected the compiler to inline that check into the callers of `sched_init`, so they could avoid the function call. Despite giving GCC some hints with `__expect` built-ins, it would not do what I wanted.

Chapter 4

Benchmarking

4.1 Web Server

The web server benchmark represents the ideal use-case for this user space threading library: a highly I/O driven application, a large number of threads and minimal inter-thread communication, with simple scheduling needs.

Three versions of the benchmarks have been developed: one using the lightweight threading library, another using pthreads, and a third purely event-driven version using a callback-based abstraction over epoll. The two threaded versions share much of the same code. They spawn a thread (lightweight or pthread) to service every request to the server. The web servers implement a minimal amount of HTTP/1.1, using an MIT-licensed HTTP-parser from Joyent[12] (also used by Node.js).

Originally I did not implement keep-alive connections, forcing the clients to use a new TCP connection for every HTTP request. This proved problematic however, as when the server closed the TCP connection (as per the HTTP specification) the socket would be left in the `TIME_WAIT` state for 60 seconds. Due to the high frequency of connections, these `TIME_WAIT` sockets would build up and eventually prevent creation of any new sockets. To solve this issue, keep-alive connections are now used instead — with the clients using a single TCP connection for all their requests, and avoiding any bottlenecks in the network stack.

The hard limits on open files and maximum threads in Linux had to be lifted beyond their default 4096 to 65536, for testing the higher numbers of simultaneous clients. Furthermore, in order to avoid limits in network bandwidth it was necessary to limit the CPU of the server processes to around 25% (using a `cgroup` CPU quota) — ensuring that the limiting factor was the CPU, and that the components being tested were the programming model and threading libraries, and not the network stack, network bandwidth or anything else.

4.1.1 Event-based web server

The event-driven version creates a thread for every logical processor on the system, each of which polls for events on an epoll file descriptor. Each epoll event has a callback and some data associated with it, which is then called when the event fires. The callback mechanism incurs some overhead compared to a pure state-machine based implementation, but the abstraction is widely used in production asynchronous I/O systems (for example, libev, Node.js, Boost ASIO and others) because it eases the burden on the programmer considerably.

The server is split into a library that provides generic routines for asynchronous I/O (`async_read`, `async_write`, and so on) and event loops, and the HTTP server logic in its own files. I have tried to keep allocations to a minimum, with the only one dynamic allocation for each connection, which should allow the event-based model to show its advantages. Reading the source code for the event-based web server should show that while it is not impossible to understand, the code is quite convoluted due to the inversion of control necessitated by the event-based model. As illustrated by the numbering in listing 4, control passes through functions in sometimes unexpected orders, and functions pass around pointers to other functions that will essentially 'continue' their work at a later time.

```
static ssize_t sendf(write_cb on_done, ...) {
    ...
    return async_send(..., on_done);                                (3)
}


static ssize_t send_http_headers(..., write_cb on_done) {
    ...
    return sendf(on_done, ...);                                     (2)
}

static void send_http_response_helper(...) {
    ...
    async_send(request->data, ..., request->on_response_done);      (4)
}

static void send_http_response(void* data, ..., write_cb on_done) {
    request->data = data;
    request->on_response_done = on_done;
    ...
    send_http_headers(..., send_http_response_helper);              (1)
}
```

Listing 4: Convoluted control flow in event-based web server code

These disadvantages can be mitigated somewhat by use of closures and anonymous functions in languages that support them (see for example Node.js code) — easing the storing and restoring of state between callbacks (i.e. 'stack ripping') — but standard C has no such support.

4.1.2 Thread-based web server

The pthread and lightweight-thread based web servers share much of the same code, with some `#ifdefs` for thread creation and I/O routines. The structure of the code is simple: one thread continually calls `accept` on a listening socket, spawning a new thread (lightweight or pthread) to handle the connection. In an effort to provide a fair comparison with the event-based version, the threaded versions also avoid dynamic allocations and allocate only on the stack. The code is simpler due to the synchronous nature, as seen below.

The thread-based web servers differ slightly from the event-based one in how they handle new clients. In the event-based web server, events occurring on the listening socket can be served by any of the handling threads (one per CPU), allowing connections to be set up and resources allocated for multiple new connections simultaneously. On the thread-based web servers, only one thread is dedicated to accepting new connections on the listening socket and setting up resources for them. The amount of operations involved in setting up a new connection are few however, so I doubt this will be particularly problematic.

```
static int sendf(int sock, ...) {
    ...
    return send_all(sock, buffer, length);                    (3)
}

static void send_http_headers(int sock, ...) {
    ...
    int result = sendf(sock, ...);                            (2)
}

static void send_http_response(int sock, void* data, ...) {
    send_http_headers(sock, ...);                             (1)

    int result = send_all(sock, data, ...);                   (4)
}
```

Listing 5: Sequential threaded web server code

4.2 Benchmarking clients

For the benchmarking clients I decided to use the `wrk` tool developed by Will Glozer[10]. It performs significantly better with large numbers of clients than the traditional Apache `ab` benchmark tool because of its use of scalable event polling mechanisms (epoll, kqueue) and its support for multithreading. I also considered 'gobench', written in Go, but it was less useful as it did not record any latency statistics. The `wrk` tool can be run independently on a number of machines to put load on the server without the client machines becoming the bottleneck.

4.3 Measurements

The measurements of particular interest are:

- Throughput (requests per second), which, when compared between each of the web servers, will give a good indication of the overhead of each of the threading libraries and programming models (assuming other bottlenecks are eliminated). Throughput is be reported by each of the client benchmark instances and then summed.
- Latency (milliseconds) — mean, maximum and standard deviation. This is interesting to compare to compare between the different programming models because of their differing modes of execution. The event-based server, and the lightweight-threads implementation with preemption disabled, will deal with a request until it is forced to switch to another due to I/O. With preemptive threading however, a request can be preempted by another at any point, as is the case with the pthreads implementation.
- Timeout rate (as a percentage of all requests attempted). Ideally no errors should occur, but with a large amount of contention between clients timeouts are possible so it is interesting to compare the servers' performance on this front.

4.4 Results

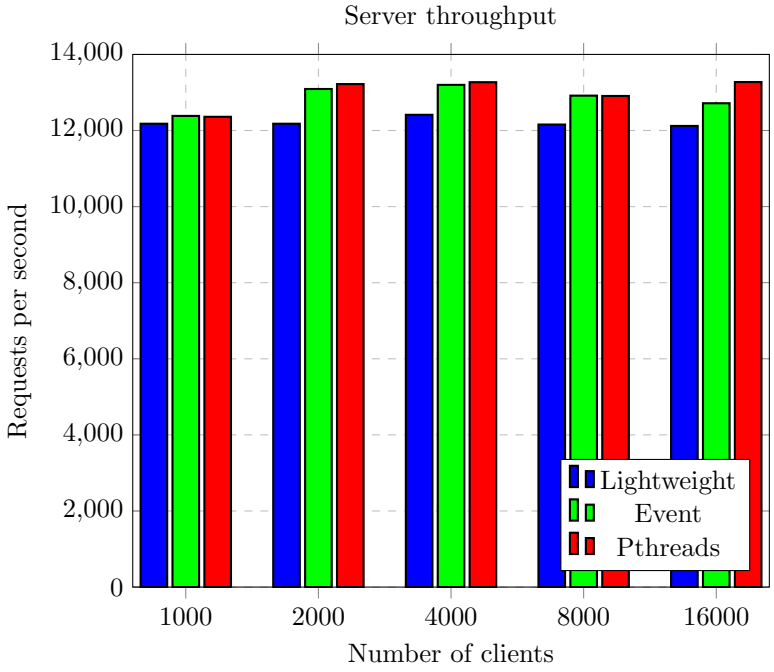
The server process, as previously mentioned, was limited to 10% CPU time to avoid 'chasing the bottleneck' — 10% CPU usage per core on the 4 core machine. The server ran on a virtual machine with the following specifications:

- Scientific Linux 7 (kernel 3.10.0-123.x86_64)
- Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz — 4 cores, no SMT
- 8GB RAM

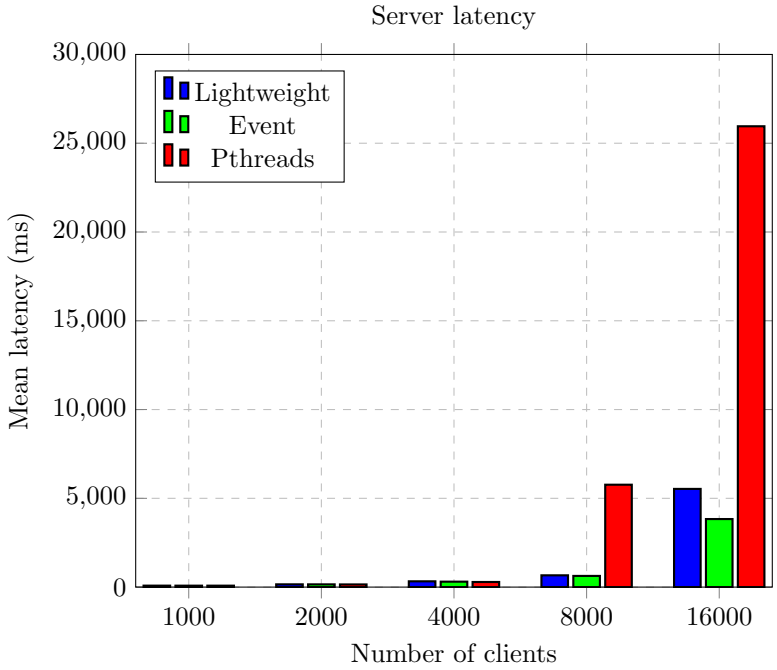
The virtual machine ran on a Microsoft hypervisor, on a machine with the following specifications:

- Microsoft Windows Server 2012
- 2x Intel Xeon E5-2620 — 6 cores (2 hardware threads each)
- 128GB DDR3 RAM

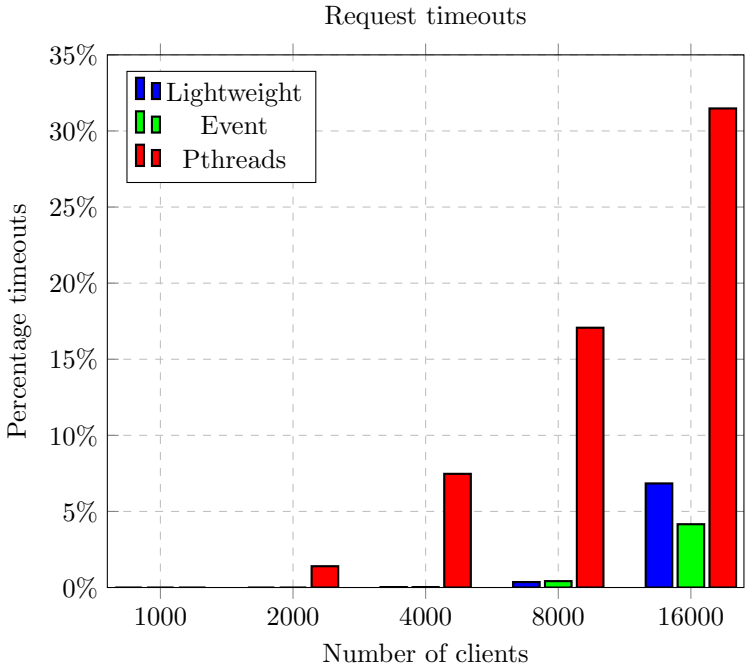
I have run the benchmarks with varying numbers of concurrent clients to see how each solution deals with increasing load. The benchmark clients were distributed over 4 different machines.



The number of requests per second is fairly constant across each number of concurrent clients. This is as I expected for the event-based server and, to a degree, the lightweight thread based server; however I expected the pthread based server to drop off in throughput due to the overhead of managing many thousands of threads. To give a more complete picture, let us examine request latency.

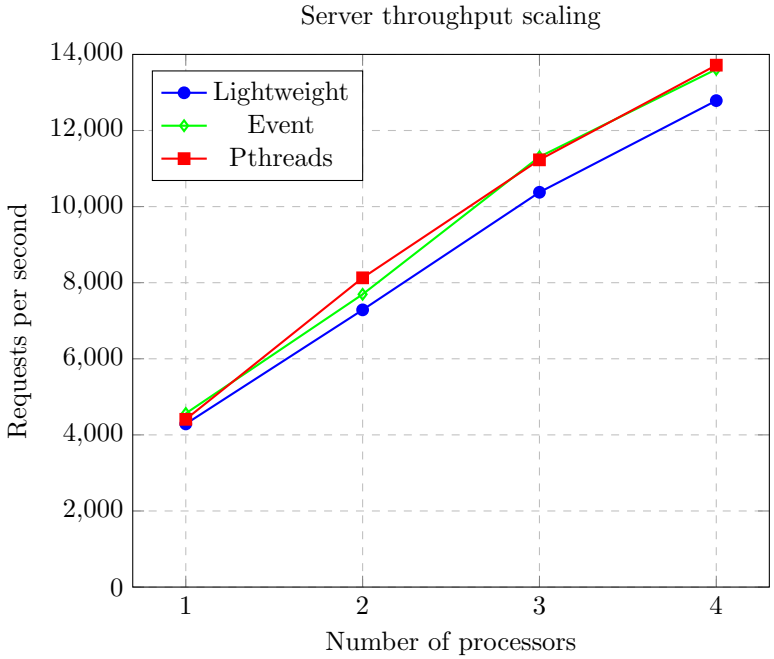


Investigating latency, however, reveals that the event-based and lightweight-thread models have a considerable advantage when it comes to request latency. At 16000 concurrent clients, the pthread-based server took on average more than 25 seconds to reply to requests. The event-based and lightweight-thread servers took around 5 seconds by comparison. The event-based server, as expected has lower overhead than the lightweight-thread solution, and so overtakes it at the higher client counts.

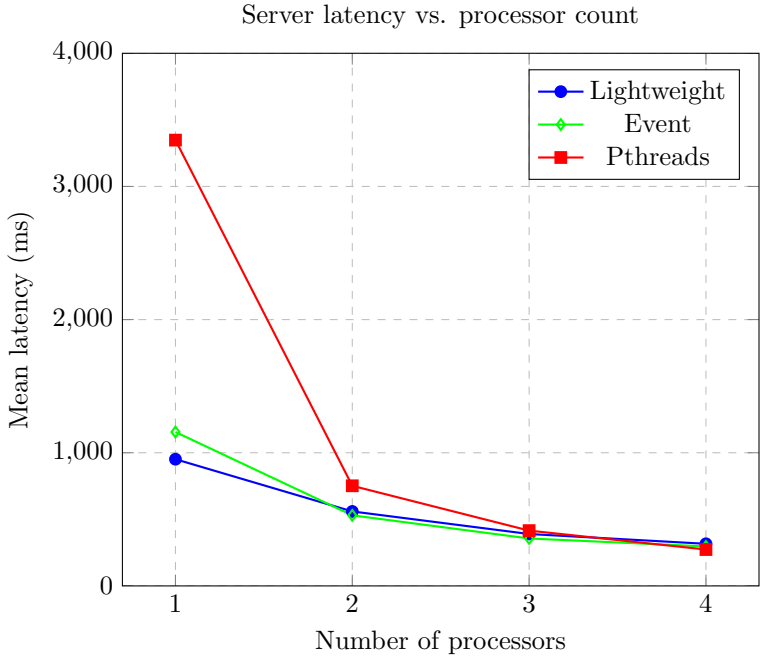


The benchmarking clients calculate timeouts based on requests that take longer than a fixed time to complete (in this case 20 seconds). This reveals that the pthreads-based server also has considerably higher variability in its latency than the other solutions. At 4000 clients, for example, the pthread-based server has an average latency similar to (and actually slightly lower) than the other two servers, but as the timeouts graph shows, the response time is less consistent than the other servers, with 7.5% of its requests taking 20 seconds or more to return.

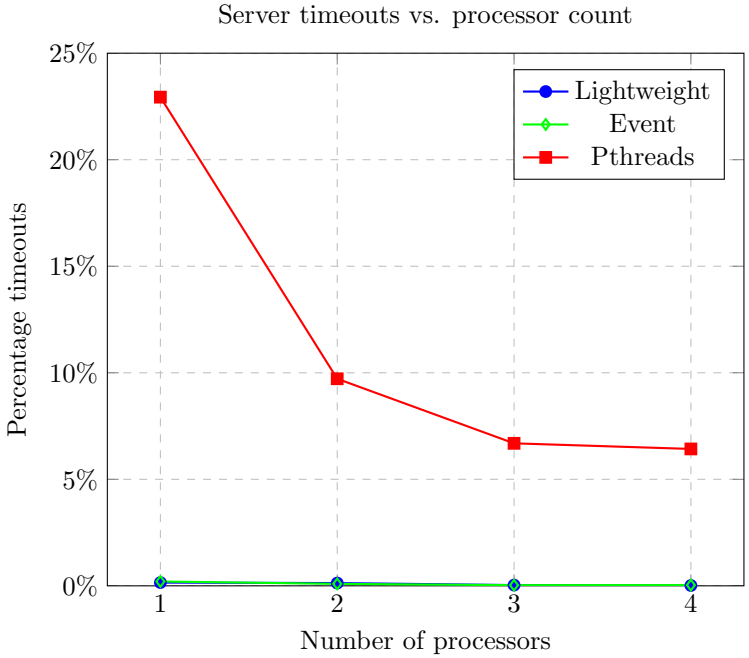
Another interesting aspect to investigate is how the different threading libraries scale with additional cores. Since the server process is running on a virtual machine I have some reservations about the results, as they may not represent how a server would scale running on physical hardware. However, since a large number of infrastructure-as-a-service providers use virtual machines to provide their hosting, such a set up is not necessarily too far removed from a 'real-world' environment.



All the servers exhibit a similar pattern of scaling, with the lightweight threaded server falling behind at higher core counts. The request throughput does not increase linearly with the core count; doubling the number of cores does not double the throughput. It would be interesting to repeat this benchmark on a server with more cores available and see if, and when, the scaling begins to taper off.



Again we see that request latency for the pthreads suffers greatly compared to the other two servers when the number of concurrent clients is high compared to the CPU resources it has available. With more CPU resources available however its average latency improves considerably.



Interestingly the timeouts of the pthread-based server do not improve moving from 3 processors to 4. Despite the mean request latency being very similar for the three servers when run on 3 and 4 processors, the pthread server still shows a problem with timeouts, indicating a high variability of latency.

Chapter 5

Future Work and Improvements

Numerous directions for future improvements to the library have been noted throughout the report. I summarise them, and a few more ideas, here.

5.1 Portability

Currently the library runs on Linux only, as it makes use some Linux-specific features such as `epoll`. As noted in other sections of the report however, all the features of the library are possible to implement on other operating systems. FreeBSD would be interesting for its `kqueue` event notification API, which provides richer functionality than `epoll`. Beyond this, the differences between the two operating systems, and indeed other UNIX-like operating systems, from the library's point of view are minor, and porting would not be difficult. The other BSD operating systems also provide `kqueue`, and Solaris provides a system called 'event ports'.

Windows is a bigger challenge as its model for scalable I/O, as mentioned in the introduction, is completion-based rather than readiness-based. This would necessitate considerable changes to the asynchronous I/O routines to support, and some changes in the scheduler too. Windows would also need an alternative implementation for preemptive scheduling, as discussed in the implementation section.

5.2 Extended non-blocking operation support

While the library provides asynchronous replacements for the common I/O and socket functions, there are more non-blocking operations that could be integrated into the event loop too. Both Linux and FreeBSD provide support for monitoring file system operations — file creation, modification, renaming and so on. Both Linux and FreeBSD also provide the `aio` interface, which is a completion-based I/O interface which also allows asynchronous I/O on files

(which readiness notifications do not, as files are always considered 'ready' for I/O), which could expand the range of operations available to perform asynchronously.

Another interesting possibility would be a function for performing and waiting on multiple asynchronous operations at the same time — perhaps for situations where the work being done does not justify creation of multiple lightweight threads, or perhaps to express more complicated communication patterns, e.g. waiting for a message on one of two channels, while also waiting for a timeout if nothing arrives.

5.3 Thread synchronisation and communication

I did not get around to implementing any thread synchronisation primitives, as they were not needed by the web server benchmark. Many applications have more complicated interthread communication needs however, and providing a basic set of primitives for communication and synchronisation is a must for them. A wait queue with a double-checked wait could provide a basic building block for other mechanisms, like mutexes and condition variables. As an alternative mechanism, message queues could be provided too, allowing threads to synchronise and communicate with each other by way of message passing.

Chapter 6

Conclusion

Implementing a user-level threading library has been an interesting experience, particularly in solving some of the trickier concurrency and synchronisation issues with the multithreaded scheduler. Some of the implementation issues — especially regarding TLS and preemptive scheduling — show that there are serious limitations to the approach of implementing threads purely in library form at user-level, without the kernel and system libraries being aware of it. These limitations might become even more apparent were the library to be used in a more complicated system, along with more complex foreign libraries that make assumptions about how threading is implemented. For this reason I more promise in the approach taken by others — such as Google's kernel-friendly user-level threads, and Barrelfish's scheduler activations — which attempt to cooperate more with the kernel, while still retaining the performance benefits of user-level scheduling.

The benchmarks did not give exactly the results I was expecting; I was expecting the pthread-based server's throughput to decrease under increasing numbers of concurrent clients due to overhead of scheduling very large numbers of threads. I was surprised to see that the throughput remained consistent. However, when examining the other factors — latency and timeout rate — the limitations of the pthreads scalability become apparent and the lightweight threading library shows its strengths, performing close to the event-based solution. Overall the benchmarks show that the library is effective for implementing simple highly concurrent server applications.

References

[1] Josh Aas. *Understanding the Linux 2.6.8.1 CPU Scheduler*. SGI Graphics, Inc. Feb. 17, 2005. URL: http://www.inf.ed.ac.uk/teaching/courses/os/coursework/lcpusched-fullpage-2x1.pdf.

[2] Thomas E Anderson et al. "Scheduler activations: Effective kernel support for the user-level management of parallelism". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 53–79.

[3] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. "A Scalable and Explicit Event Delivery Mechanism for UNIX". In: *Usenix Annual Technical Conference*. 1999, pp. 253–265. URL: http://www.cs.rice.edu/~druschel/usenix99event.ps.gz.

[4] Andrew Baumann et al. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 29–44.

[5] Robert D Blumofe and Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[6] Neil Deshpande, Erica Sponsler, and Nathaniel Weiss. "Analysis of the Go runtime scheduler". In: (2012).

[7] Julian Elischer. *Strawman proposal: making libthr default thread implementation?* July 4, 2006. URL: http://lists.freebsd.org/pipermail/freebsd-threads/2006-July/003592.html.

[8] *epoll — I/O event notification facility*. URL: http://man7.org/linux/man-pages/man7/epoll.7.html.

[9] Apache Foundation. *Multi-Processing Modules (MPMs)*. 2015. URL: https://web.archive.org/web/20150309073644/http://httpd.apache.org/docs/2.4/mpm.html (visited on 04/08/2015).

[10] Will Glozer. *wrk - a HTTP benchmarking tool*. 2015. URL: https://web.archive.org/web/20150204203944/https://github.com/wg/wrk.

[11] ISO ISO. "IEC 9899: 1999: Programming languages C". In: *International Organization for Standardization* (1999), pp. 243–245. URL: http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf.

[12] Joyent. *HTTP request/response parser for C*. 2015. URL: https://github.com/joyent/http-parser.

[13] Dan Kegel. *The C10K problem*. URL: http://www.kegel.com/c10k.html.

[14] Jan Kneschke. *Benchmark — Static file throughput*. 2004. URL: https://web.archive.org/web/20041230034743/http://www.lighttpd.net/benchmark/.

[15] Jonathan Lemon. "Kqueue: A Generic and Scalable Event Notification Facility". In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 141–153. ISBN: 1-880446-10-3. URL: http://dl.acm.org/citation.cfm?id=647054.715764.

[16] Kenneth Lundin. "Inside the Erlang VM with focus on SMP". In: *Erlang User Conference, Stockholm*. Ericsson AB. Nov. 13, 2008. URL: http://www.erlang.org/euc/08/euc_smp.pdf.

[17] Mindaugas Rasiukevicius. *Thread scheduling and related interfaces in NetBSD 5.0*. The NetBSD Project. May 4, 2009. URL: http://www.netbsd.org/~rmind/pub/netbsd_5_scheduling_apis.pdf.

[18] The Open Group. *makecontext, swapcontext - manipulate user contexts*. 2004. URL: http://pubs.opengroup.org/onlinepubs/009695399/functions/makecontext.html.

[19] Paul Turner. "User-level threads... with threads." In: *Linux Plumbers Conference*. 2013. URL: http://www.linuxplumbersconf.org/2013/ocw//system/presentations/1653/original/LPC%20-%20User%20Threading.pdf.

[20] George Varghese and Anthony Lauck. "Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility". In: *IEEE/ACM Transactions on Networking* 5.6 (1997), pp. 824–834.

[21] Various. *net: use non-blocking syscall.Syscall for non-blocking calls*. 2012. URL: https://code.google.com/p/go/issues/detail?id=3412.

[22] John Vert. *Writing Scalable Applications for Windows NT*. 6, 1995. URL: https://msdn.microsoft.com/enIEEE/ACM%20Transactions%20on-us/library/ms810434.aspx.

Appendix A

User Manual

A.1 Building

The code can be checked out from the GitHub repository (currently private):

```
git clone https://github.com/hamishm/shproject.git

cd shproject

# Checks out http-parser repository used by web server benchmark
git submodule update --init --recursive
```

If building from the included source code, it will be necessary to check out the http-parser repository manually. From the root of the project source directory, run:

```
git clone https://github.com/joyent/http-parser.git bench/http-parser
```

In case of an API change in a later version of `http-parser` that breaks the build, a known working revision is `53063b7`.

The code can be built using `make`. There are a couple of optional features that can be enabled with environment variables:

```
# Build non-debug and without preemption
make

# Build with preemption enabled
make PREEMPTION=1

# Build as a shared library (will force PIC)
make SHARED=1

# Build with debugging information
make DEBUG=1
```

This will produce a static library `liblwthread.a`, and optionally a shared library `liblwthread.so`, with which you can link your program. On Linux if you use the static library you must also link with `librt`, and build using the `-pthread` option. For example:

```
gcc mycode.c -llwthread -lrt -pthread
```

A.2 API

```
/*
 * Create a new lightweight thread which will execute the given function, and
 * pass the given argument as its first parameter.
 * Returns a pointer to the new thread on success, or NULL on error.
 * The lightweight thread will be automatically deallocated when it exits.
 */
struct coroutine* sched_new_coroutine(void* (*start)(void*), void* arg);

/*
 * Read up to length bytes from the file descriptor into the given buffer.
 *
 * Returns the number of bytes read, or -1 if an error occurred.
 * Consult sched_errno for the error.
 */
ssize_t async_read(int fd, void* buf, size_t length);

/*
 * Write up to length bytes to the file descriptor from the given buffer.
 *
 * Returns the number of bytes written, or -1 if an error occurred.
 * Consult sched_errno for the error.
 */
ssize_t async_write(int fd, void* buf, size_t length);

/*
 * Accept a pending connection from the given listening socket. If address and
 * length are non-NULL, they will be filled in the with the peer's address.
 *
 * Returns the file descriptor of the accepted socket on success, or -1 if
 * an error occurred. Consult sched_errno for the error.
 */
int async_accept(int socket, struct sockaddr* address, socklen_t* length);

/*
 * Connect the given socket to the given address.
 * Returns 0 on success and -1 on error. Consult sched_errno for the error.
 */
int async_connect(int socket, const struct sockaddr* address, socklen_t length);
```

```c
/*
 * Sends up to length bytes from the given buffer to the given socket.
 * The flags argument is passed to the operating system's send call. Consult
 * your operating system's documentation for possible values.
 *
 * Returns the number of bytes sent on success, or -1 if an error occurred.
 * Consult sched_errno for the error.
 */
ssize_t async_send(int socket, const void* buffer, size_t length, int flags);

/*
 * Receive up to length bytes into the given buffer from the given socket.
 * The flags argument is passed to the operating system's send call. Consult
 * your operating systems' documentation for possible values.
 *
 * Returns the number of bytes received on success, or -1 if an error occurred.
 * Consult sched_errno for the error.
 */
ssize_t async_recv(int socket, void* buffer, size_t length, int flags);

/*
 * Sleep the current lightweight thread for the given number of milliseconds.
 */
void async_sleep_relative(long millisecs);

/*
 * Sleep until the given absolute time. The timespec struct is defined in the
 * POSIX standard.
 */
void async_sleep_absolute(const struct timespec* time);
```