Green Threads

User level threading for a Library OS

Vinay Shetty, Michael Ferdman
Computer Science, Stony Brook University
{vishetty, mferdman}@cs.stonybrook.edu

Fall 2013

1 Introduction

Library operating systems have long been researched to study the impact of pushing many functionalities out of the kernel and into userspace, for the purposes of security, compatibility and performance. The Drawbridge[1] project demonstrated that a commercial, large operating system (OS) like Windows can be refactored into a Library OS supporting standalone apps, providing access to low-level resources using only a small set of ABIs. The Graphene project generalizes this idea to the Linux OS, and further adds secure multiprocessing abstractions and new classes of applications to the mix.

An interesting artifact of this design decision is the clear demarcation between the unprivileged, userspace component (the library OS) and the privileged, kernel component. This well-defined boundary lends itself well to execution on systems with heterogeneous cores[2], where different cores have different capabilities for varying energy and performance characteristics. In our case, we specifically look at cores which support different protection domains. For example, on the Intel architecture, simple cores will support only ring 3, while complex cores support ring 0 as well. The fact that a library OS limits most of the unprivileged OS code to userspace, with the privileged code being accessed via a narrow API, makes it very well suited for experimentation with heterogeneous multi-core systems. Thus, most of the OS, that is the userspace component, can be executed on a simple core and the kernel component using the complex core.

In traditional OSes, user level threading is one model of multi-threading which is employed for its performance benefits. In contrast to kernel level threads, they are light-weight since most of the thread management, including context switching and thread scheduling is done in user space without involving the kernel. Moreover, the application can be in control of the threading, instead of being limited by the kernel. However, this approach is not without its disadvantages; the most prominent ones being: user level threads cannot simultaneously run on multiple processors and no user level thread can proceed if one of them is blocked performing a system call[3].

Despite the suitability of the Graphene Library OS for our purpose, threading is something that is still kernel based. In this project, we aim to support multi-threaded applications on heterogeneous systems by moving the responsibility of creating and managing threads from the kernel to userspace as provided by a user level threading implementation.

2 Objective

This work is an attempt to introduce User Level Threading (ULT) into a library OS with a view towards enabling its execution on systems with heterogeneous cores, specifically those that support different protection domains. As a first step towards this, we aim to provide a light-weight implementation of ULT in Graphene to be tested on Linux. This way we ensure that a multi-threaded application does not have to make system calls into the kernel for thread creation and management purposes. In order to accomplish this, we adhere to a constraint that user threads will not

execute any system calls. This is because system calls execute in privileged mode and our aim is to restrict threading to a simple core, with system calls being executed on the complex core.

A key component of this effort is the implementation of non-blocking system calls in order to ensure that other schedulable user threads are not blocked by a system call being executed on behalf of a single user thread.

3 Graphene

Graphene library OS supports execution of unmodified applications and associated libraries written for a Linux OS; it provides security through isolation of different processes while also supporting inter-process communication. To achieve this isolation, the bulk of OS functionality is pushed out of the kernel into a layer above which is the actual "library" OS. This layer consists of two parts: a platform-independent part compiled into a shared object called libLinux.so; and a platform-specific part called Platform Adaptation Layer (abbreviated as PAL), compiled into another shared library called libpal.so.

The libLinux library mainly houses implementation of system calls which are used by the standard C library(libc) wrapper calls, which in turn are invoked by the application, using a small set of lower-level calls defined by the Drawbridge ABI. The re-routing of the system calls is achieved by modifying the C library(glibc for Linux), replacing all direct system calls into the kernel, with calls into the libLinux library. In addition to system calls, this library re-implements many of the Linux kernel functionality required by applications.

The PAL library implements the Drawbridge ABI, a minimal interface exposing low-level functionality which is managed by the Host OS. The ABI(also referred to as PAL calls) does so using Linux system calls, thereby directly interacting with Host kernel. This delineation of responsibilities between the Library OS and Host OS not only aids in isolation but also allows independent development of each without having to worry about compatibility, thanks to the common ABI.

4 Green threads AKA User level threads

Graphene provides multi-threading via kernel-based threads. The libLinux library implements a platform-agnostic threading management layer. This implementation is backed by kernel level threads, implemented in the PAL layer, created using the clone system call. For the most part, we depend on the existing libLinux threading implementation to do the heavy lifting, whereas our green thread implementation simply acts as a replacement for the lower-level threading related PAL ABI calls. However, an important thing to note is that we do not implement threads in the PAL layer. While it would be convenient to relegate changes to the PAL layer leaving the libLinux layer undisturbed, we seek to provide a platform-independent threading implementation and hence choose to implement our version of green threads in libLinux. Note that we use the term green threads and user threads interchangeably.

While there are several opensource ULT libraries available, they don't exactly fit our particular requirements. Specifically, most threading implementations either provide a user-level threading implemenation of the Pthread specification[4] or provide threading APIs of their own[5]. For our purpose, we don't want to modify the existing Pthread implementation provided by glibc or rewrite the whole libLinux threading library which backs the glibc impelmentation. Instead, we choose to provide a minimally intrusive implementation which avoids calling into the kernel everytime for thread management. Optionally, we allow switching between user-level and kernel thread implementations as a compile-time option.

With regards to the non-blocking system call support, the ULT libraries which provide such support do so using a single threaded event-based framework[6] aimed at avoiding synchronous I/O calls. Since we aim to avoid any kind of blocking system call in general, we choose to implement the same using an internal "system call(syscall) handler" kernel thread, which impersonates a user thread on whose behalf it is currently executing. This design also scales well by allowing

addition of sycall handler kernel threads in future.

4.1 Core threading library

The core green thread implementation consists of thread creation, management and scheduling. Currently, we employ an M:1 threading model where M user threads are served by, effectively, a single kernel thread. The thread operations like create, yield, resume and exit are one-to-one replacements for their libLinux thread counterparts which exist in Graphene. Thread switching is accomplished by saving and restoring the register state as implemented by the standard C library operations, longjmp and setjmp. However, this is platform-specific assembly code and we only support x86_64 architecture for now. But this can be easily extended to support other platforms, similar to platform specific implementations already implemented in glibc. We use the co-operative thread scheduling model, that is, we use round-robin like order without a fixed timeslice. In other words, we schedule the next available thread and let it run until it yields control or gets suspended. It is not unusual for user threads to employ co-operative scheduling since all threads take up the timeslice of the same process as allocated by the OS' scheduler. Pre-emptive scheduling could be implemented, if required, using the alarm implementation present in Graphene.

4.2 Userspace Futex implementation

In Graphene, all synchronization primitives in libLinux are implemented based on PAL synchronization APIs. The Linux PAL implementation, in turn, uses the single futex system call to provide wait and wake facilities for a kernel thread. Both the libLinux library and the green thread implementation use the aforementioned synchronization primitives. For kernel threading implementation, the futex system call is needed to access kernel threads in order to suspend and resume their execution. However, for a user thread implementation, we can avoid making the futex system call required by synchronization in Graphene. Thus, we implemented user level futex queues which hold the user threads waiting on locks. We do this for two reasons: first, similar to moving threading from kernel to userspace, it makes sense to move the futex implementation to userspace as well, since we then have access to user level threads; second, locking is thread-specific and the futex call execution cannot be offloaded to another thread as with our non-blocking system call handling support.

There is, however, a catch. In addition to libLinux and green threads using the PAL synchronization mechanism, the PAL code also uses the same APIs for synchronization. Since all libLinux locking primitives end up calling into the PAL layer and the PAL code also currently uses the same underlying locking implementation, we needed a single solution for a futex implementation. In order to have a single unified futex implementation, yet still have access to green threads which exist in libLinux, we provide a callback from PAL to libLinux. This callback ensures that the futex implementation can exist at the libLinux layer and have access to green threads. We acknowledge that this violates the design principles of Graphene, but is meant only to serve as a temporary measure. See the Future work section for an alternative implementation.

The user level futex implementation maintains a set of futex-specific queues, with each entry being a thread waiting on the corresponding futex. Every waiting thread on the queue is suspended and hence not schedulable by the green thread scheduler. We use MCS(Mellor-Crummey and Scott) mutual exclusion lock to protect these queues from concurrent accesses.

4.3 Signal Handling

Since POSIX implementations support signals on a per-thread basis, we needed to support similar signal handling semantics for our work as well. Fortunately, the libLinux library has a comprehensive signal handling implementation which installs generic signal handlers in the kernel at system startup, and then demultiplexes signals across user threads based on signal masks maintained for each libLinux thread. Only a single modification was made to adapt the signal handling code for green threads. Note that since we depend on libLinux signal implementation, the signal handling semantics are therefore not decided by our green thread implementation.

4.4 Non-blocking system call support

In Graphene, all system calls to the host kernel are guaranteed to be invoked from the PAL layer. This isolation of system calls to the PAL layer lends itself well to the execution of PAL calls in a separate kernel thread. We use this secondary handler thread to allow the other schedulable user threads to make progress.

The syscall handling thread is a kernel thread created with the sole purpose of executing PAL calls (and consequently, system calls) in a separate thread. The handler thread executes a loop, dequeueing requests to execute PAL calls on behalf of each of the user threads. It also wakes up the main thread if it has gone to sleep when there are no schedulable user threads. Whenever the user thread is queued to be executed by the handler thread, we ensure that it cannot be scheduled to execute in the main thread. This technique ensures that no system calls are made by a user thread (with one exception) and, more importantly, no user thread which can be scheduled is blocked due to another user thread executing its system call.

To implement this, we provide wrappers for each of the PAL calls that exist today in Graphene. When green threads are enabled, the libLinux library automatically uses these wrappers to make PAL calls. We ensure that these wrappers are only visible to the libLinux layer and do not affect the PAL layer itself. The function of the wrappers is to conditionally enqueue the reference to the user thread to be impersonated by the handler thread and pass control from and back to the invoking user thread. It also wakes up the system call handler thread if it has gone to sleep.

An alternative mechanism would be to enqueue the PAL call and associated parameters to be dequeued by the handler thread and invoked. However, the PAL calls do not have a fixed number of arguments like a system call. Also, checking the types of the arguments and setting up the stack for the PAL call seemed to be more difficult to implement than the currently adopted mechanism.

5 Challenges

5.1 Handling synchronization between user threads and kernel threads

One consequence of the decision to implement user level futex queues for green threads is the difficulty of handling kernel threads using the same mechanism. Essentially, in its current form, the user level futex implementation does not handle suspending and waking up a kernel thread. Currently, the only interaction between user threads and kernel threads under consideration is the interaction between a user thread and a syscall handler thread. For now, we carefully avoid making calls to libLinux synchronization APIs from within the syscall handler thread. Instead, we use MCS locks within the syscall handler to avoid suspending the thread. However, there are other internal kernel threads which are spawned by the libLinux library which will also need to be considered, while providing a cleaner solution for this problem. See the Future work section for once such possible solution.

5.2 Stack setup by the syscall handler thread

Currently, the syscall handler thread executes the enqueued PAL calls on behalf of the green threads sequentially. During the impersonation of a green thread, the syscall handler thread executes in its own stack from the point where the green thread suspended itself. In order to do this correctly, the contents of the user thread stack needs to be copied correctly to the syscall handler's stack. However, we were faced with the issue of dealing with adjusting frame base pointer values stored in the stack. In order to avoid this complexity, the code has been carefully designed such that the initial stack frame of the wrapper function is shared between the user thread and the handler thread while the rest of the successive frames are separate. As a consequence, this results in a scenario where the return values of the PAL call need not be copied back to the original user thread stack. However, this does mean that any modifications to both the wrapper function prologue and epilogue, along with the stack setup done by the syscall handler thread should

be given careful consideration. Also, this code has only been tested with the gcc compiler and on a 64-bit Linux installation.

5.3 Merging with upstream changes

Graphene is a project under active development and so absorbing those changes and keeping up with design changes were a part of this effort. However, it is to be noted that the changes were not conflicting with our green thread implementation and hence, merging code bases was a relatively easy task. In future, when the green thread project gets merged into the main Graphene project, such modifications should not be a cause for concern.

6 Caveats and Limitations

We use co-operative scheduling, not pre-emptive. This is something user programs need to be aware (can result in copious use of pthread_yield). Also, thread cancellation is not supported. Therefore, support for the pthread_cancel routine will need to implemented for green threads.

As noted earlier, there are some exceptions to the "no system calls from user thread" rule. Specifically, the system call to set the base address for thread specific private data, arch_prctl, is called once during every context switch for a thread. Also, in the current implementation, the main thread suspends itself by using the nanosleep system call and wakes up a suspended syscall handler thread using the tgkill system call which is used to send a signal to another kernel thread. Both these system calls can be replaced by making the threads busy-wait on a flag.

As discussed previously, the user level futex implementation necessitates careful handling of synchronization between user threads and kernel threads. Therefore, for now, care must be taken to ensure that any interactions between user and kernel threads should be analysed for any synchorization related issues.

While all attempts have been made to integrate the green thread implementation seamlessly with Graphene. There could be areas which might not have been investigated for its impact on or a possible change in behaviour due to green threads. For example, Graphene migration is not supported, or in other words, not something that has been looked at.

7 Evaluation

To evaluate the correctness of our implementation we write unit testcases to verify each component of our implementation. Most of these tests are either based on existing multithreaded cases existing in the Graphene testsuite. For cases where no tests existed previously, separate tests were written. Mainly the userlevel futex implementation and signal handling was verified with additional tests. All tests have suffix of "gthread" in their name.

8 Future Work

8.1 Avoid system calls for Thread Local Storage

As mentioned previously, the system call to set the base address for the thread specific data, arch_prctl, violates our invariant of not making system calls from a user thread's context. There are a couple of avenues to be explored in search of a possible solution. First, the latest Intel processor provide instructions, specifically, WRFSBASE, in order to set the base address. However, this would possibly require kernel modification or configuration to enable this feature and is Intel-specific along with being available only on latest processors.

An alternative mechanism would be to keep the base address the same for all threads, while moving out the current thread's thread control block(tcb) and copying in the next thread's tcb. However, creating a copy introduces the problem of maintaining a consistent copy. And inconsistency is possible, because of two access paths, first one implicitly through fs register, the second one is through anyone accessing the struct pthread pointer for a given thread, given by its pthread_t variable.

As a complement to this, we need a mechanism to verify system calls are not being made, except for the ones we know of, when executing a user thread.

8.2 Separate libLinux and PAL locking implementations to avoid libLinux callback mechanism in PAL layer

As discussed earlier, the dependency of the libLinux synchronization implementation on the PAL synchronization mechanism caused the introduction of a callback mechanism from PAL to libLinux in order to provide a single userspace futex implementation. However, a more suitable approach to this would be to separate both the synchronization mechanisms, with the userspace implementation backed by the userlevel futex and the PAL synchronization layer backed by the futex system call. While this would cause duplication of common code, it would allow for clean separation of concerns with locking for user threads not extending beyond the libLinux layer and all PAL code still continuing to successfully use the kernel synchronization mechanism.

8.3 Handle synchronization among user threads and kernel threads

A mechanism needs to be found that will allow for transparent handling of synchronization between user threads and kernel threads. This means that the existing libLinux locking APIs should work automatically for both user threads and kernel threads like the syscall handler. Ofcourse, this enhancement should also work well with separate locking mechanisms idea described above.

8.4 Handle thread interruption and system call restart

Currently, we assume successful completion of all system calls executed via a PAL call by the syscall handler thread. However, in case this execution is interrupted for any reason, re-execution is not being handled currently.

8.5 Handle thread sleep and resume

As of now, a sleep system call gets handled by the syscall handler thread like any other system call and a thread resume call simply wakes up a suspended user thread and puts it back on the scheduler's queue. However, this could be optimized by ensuring that the syscall handler thread does not sleep on behalf of a user thread and a resume ensures that if a user thread is waiting to be served by the syscall handler thread, it gets evicted from the handler's queue and scheduled again. However, the semantics of system call interruption caused due to the resume should be handled correctly.

8.6 Extensive testing with full-fledged applications

The existing testcases for green threads are either clones of multi-threading tests for Graphene or unit tests for the new features implemented for green threads. More extensive testing in terms of multi-threaded benchmarks and applications need to be executed to flush out all bugs. A good start would be to test an already existing multi-threaded application in the Graphene test suite, so that we have a means of comparison between the kernel thread and green thread implementation.

8.7 Improvements for green threads

While parallelism among user threads is not usually found in ULT implementations, it is definitely possible to schedule threads in parallel on multiprocessor system. While we have not tested it yet, the green thread implementation already uses locking to protect its data structures and as such should be able to support parallel execution. However, thorough testing would be required to validate this. Also, the context switching code is specific to processor architecture and not portable yet. Currently, we support the x86 64 bit architecture only. Finally, while the main impetus behind this work is the enabling of Graphene to execute on heterogeneous cores, this project can also be used to analyse whether it makes more sense to do threading in user or kernel space in a library OS.

9 Acknowledgements

I would like to thank Chia-Che and Prof. Donald Porter for their help with any queries I had regarding Graphene and Prof. Mike Ferdman for

his able guidance without which this work would not have been possible.

10 References

[1] D.E. Porter, S. Boyd-Wikcizer, J. Howell, R. Olinsky, and G. Hunt. *Rethinking the library OS from the top down.* In ASPLOS, pp. 291-304, 2011.
`http://research.microsoft.com/pubs/141071/asplos2011-drawbridge.pdf`

[2] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, *Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction,* in Proc. of the 36th Annual International Symposium on Microarchitecture , pp. 81-92, 2003
`http://dl.acm.org/citation.cfm?id=956569`

[3] *Green vs. Native threads*
`http://c2.com/cgi/wiki?GreenVsNativeThreads`

[4] *GNU Portable threads*
`http://www.gnu.org/software/pth/pth-manual.html`

[5] *Qthreads*
`http://www.cs.sandia.gov/qthreads/`

[6] Wikipedia page for Asynchronous I/O
`http://en.wikipedia.org/wiki/Asynchronous_I/O`