A Proposal for an Efficient User-Space Scheduler

Nikhil Benesch
nikhil.benesch@gmail.com

1 Introduction

A survey of L3- and L4-based microkernels over twenty years reveals all implementations “have a scheduler in the kernel, which implements a particular scheduling policy” [1]. Yet this is a blatant but widely-accepted violation of the microkernel policy-mechanism separation principle: the kernel should be responsible only for providing mechanisms so that all policy decisions can rest with userspace components.

Suppose we accept the stated benefits of microkernel architecture: that microkernels improve reliability, security, and flexibility through modularity and the principle of least privilege. I claim, then, that removing the scheduler furthers these goals. Clearly, a userspace scheduler improves flexibility: your users are not bound by whichever scheduling algorithm you happened to implement in the kernel. How userspace schedulers improve software reliability is less clear, though I claim that reliability is improved through:

- allowing easy creation of debugging schedulers, like implementations with a deterministic or deliberately adversarial scheduling algorithm,
- implicit support for dynamically swapping schedulers at runtime to adapt to changing workloads, and
- enabling isolated testing of each scheduler, enabling deeper formal modeling. The current seL4 abstract model only ensures that the seL4 scheduler selects a runnable thread to run.

Userspace schedulers likely yield little improvement to a system’s security, except that they remove yet another chunk of code from operating in a trusted privilege level and thus reduce the impact of scheduler bugs to a denial of service.

I additionally cite L4 inventor Liedtke’s minimality principle as moral obligation, as I have yet to see convincing evidence that a userspace scheduler prevents the implementation of a functional system.

2 Goals

Why is efficient userspace scheduling so difficult?

This is the fundamental question this project seeks to answer. The literature we’ve read in this class dismisses the notion of a user-level scheduler as yet-impossible or unworthy. Elphinstone and Heiser note in their 2013 survey of L4 microkernels that “to date, no one has come up with a truly general in-kernel scheduler or a workable mechanism which would delegate all scheduling policy to user-level without imposing high overhead” [1]. Consider the challenge accepted.

The ideal deliverable is a scheduler for seL4, with policy implemented entirely in userland, capable of performance within an order of magnitude of stock seL4. Formal verification is outside the scope of this project, but the ideal deliverable does not preclude future work from re-verifying the timeliness guarantees that stock seL4 currently satisfies. I’d like whatever scheduling mechanisms the kernel provides to be sufficiently general that all reasonable algorithms are feasible, including those with real-time guarantees.

This requires not only answering the question of why userspace scheduling is difficult, but developing a satisfactory solution that is order(s) of magnitude faster than the naive solution. No small task.

I note that the impact of this deliverable is rather hard to quantify. A userspace scheduler will not itself catch
any bugs; other processes will run slower, not faster. But I think the goal is still worthwhile.

If disaster strikes, a minimal deliverable for this project is rather disappointing: several forks of seL4 with schedulers too inefficient to be useful, some benchmarks to prove this, and a survey of the existing research into semi-successful techniques to extract the scheduler from the kernel.

I’m worried this is a problem that remains unsolved not for lack of interest, but for its inherent infeasibility. Interestingly, I’m also worried about the converse. Additional digging has yielded more literature upon the subject than I expected, including several strategies for implementing userspace schedulers on top of seL4! (See “Related Work” below.)

3 Technologies

I plan to use only seL4 and the recommended toolchain, a cross-compiling GCC on Debian capable of building a QEMU-compatible image. The tools used to formally verify seL4 will largely be ignored.

4 Related Work

Perhaps most relevant is a paper entitled “Towards Effective User-Controlled Scheduling for Microkernel-Based Systems” by Jan Stoess [3]. Stoess describes an impressive mechanism that relies largely on user processes implicitly or explicitly yielding time to processes it communicates with via IPC semantics. The architecture imposes only a 10% overhead, but with an apparently substantial loss of generality of workloads that the scheduler can support. I’d like to investigate this more this week—the caveats of his scheme are far from clear to me.

Stoess notes several additional papers on the subject. Worth investigating are two-level scheduling schemes that place “dispatchers” in the kernel that schedule the user space “schedulers,” although conceptually this seems to exacerbate the problem. Also worth investigating is the CPU inheritance scheduling, which is similar to Stoess’s approach.

Additionally cited are several papers on the implementation of userspace-like or real-time schedulers in several systems, including seL4.

I’ve omitted the citations for the rest of the papers in the interest of time. I think Stoess [3] is a good candidate for reading in class, though not particularly related to systems debugging. I also found the more topical “seL4: Formal Verification of an Operating-System Kernel” to be a fascinating and approachable analysis of applying formal verification to a project of a size I wouldn’t have thought possible.

5 One-Week Deliverable

First, I need a working seL4 toolchain. I’ve successfully set up a Debian virtual machine with a provisioning script capable of reliably installing the seL4 toolchain and compiling a kernel that can pass its internal test suite.

Next, I’ll establish baseline scheduler performance across three platforms: native seL4, Linux on top of seL4, and native Linux. Linux has a useful userland, so I’ll follow Stoess’s strategy and compare preemption frequency and interrupt latency while running the Apache benchmarking tool ab and netperf concurrently. When running ab against a local HTTP server, this workload requires frequent context switches. Testing seL4 directly is a bit more challenging, as seL4 doesn’t ship with POSIX compatibility or a useful userland. I’ll again follow Stoess’s lead and utilize a microbenchmark that plays IPC ping-pong between several processes, measuring the round-trip time for each IPC.

Time permitting, I’ll rip out the existing round-robin scheduler in the seL4 kernel and naively implement it in userspace. While I haven’t yet investigated the code, I imagine this to be something along the lines of replacing every call in the kernel to schedule with a context switch to the userland scheduler with a rather large message containing all the data the kernel previously used to make its scheduling decision.

In the process, I hope to familiarize myself with the seL4 architecture and codebase. To be successful, I need a better understanding of the intricacies of seL4 IPC, which prior research suggests are fundamentally tied to scheduler implementation strategies and performance. I’d also like to do a thorough scan of the related work to ensure my approach is novel.

I’ll be keeping track of my progress in the schedul4 directory of the usual Git repository, https://github.com/benesch/cs260r. Available so far is this source code of this proposal and a Vagrant configuration for automatically creating a Debian seL4 build environment.

6 Collaboration Plan

The advantage of a uniprocessor system: no synchronization nightmares. The disadvantage: no opportunities for parallelism.

Eddie, if possible, I’d like to check in with you and bounce ideas off you perhaps more frequently than other groups since I’ll be working alone.

7 Remaining Questions

1. How much time should I spend reviewing prior literature—especially non-seL4 literature—before
embarking on my own solution?

2. Which implementation strategy seems most likely to be successful?

3. What’s the minimum deliverable that you, Eddie, would be satisfied with? Can we adjust the minimum deliverable to seem like less of a failure? Or, alternatively, do you think I’ll have a good sense of whether this project is within my capabilities by the end of this first week?

References

