# Lightweight Preemptible Functions

A thesis

**Sol Boucher**

CMU-CS-22-101
March 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Thesis committee:**
David G. Andersen, *chair*
Adam Belay
Michael Kaminsky
Brandon Lucia

*Submitted in partial fulfillment of the requirements*
*for the degree of Doctor of Philosophy*



This research was sponsored by the Pradeep Sindhu Fellowship, VMWare, Google, Intel Science and Technology Center for Cloud Computing, and the National Science Foundation under grant number 1700521. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

**Keywords:** programming primitives, operating systems, concurrency, preemption, isolation, reentrancy, dynamic linking, timer signals, asynchronous cancellation, preemptive user threads

# Abstract

We introduce novel programming abstractions for isolation of both time and memory. They operate at finer granularity than traditional primitives, supporting preemption at sub-millisecond timescales and tasks defined at the level of a function call. This resolution enables new functionality for application programmers, including users of unmanaged systems programming languages, all without requiring changes to the existing systems stack. Despite being concurrency abstractions, they employ synchronous invocation to allow application programmers to make their own scheduling decisions. However, we found that they compose naturally with existing concurrency abstractions centered around asynchronous background work, such as threads and futures. We demonstrated how such composition can enable asynchronous cancellation of threads and the implementation of preemptive thread libraries in userland, both regarded for decades as challenging problems.

# Table of Contents

# List of Figures

# List of Tables

# List of Listings

# Acknowledgments

I wish to thank my adviser, Dave Andersen, for his unwavering support and for never giving up on me despite extended periods of little progress. I especially appreciate his encouragement to take vacations and backing of my teaching, even when it meant taking time off from research.

A doctorate can be an emotional roller coaster ride, and I owe many people thanks for keeping me from derailing on the tricky corners (and corner cases). The good doctors Ben Blum and Dominic Chen, both of whom were students in the program when I started, inspired me with their intense systems projects. Their perseverance in conducting challenging low-level research convinced me that operating systems research is far from "dead," as some professors had tried to tell me when I was seeking an adviser. On more than one occasion, Dominic rescued me from debugging funks when I spent weeks unsuccessfully searching for horrible bugs; I am also forever indebted to him for introducing me to the rr reverse debugger. Ben's earlier work on Rust was one of the factors that led me to learn the programming language. Another colleague, Gabriele Farina, challenged me to become fluent in it and encouraged me to join him at Rust conferences and attend the Pittsburgh Rust coffee meetup. At the latter, I met Holden Marcsisin, then a star local high school student who helped me reason about safe unstructured jumps in the context of Rust. Dave's (now) former student Anuj Kalia was instrumental in my research pivot toward the more fulfilling area of novel programming primitives: his suggestion during a meeting about our microservices work that we could leverage dynamic linking to address the nonreentrancy problems associated with asynchronous cancellation led directly to my work on selective relinking, which made everything else in this dissertation possible. And I owe a great debt to my many friends in the program who kept things fun (and me sane) through the hard times with regular board games and other social events, bicycle rides and trips, and the occasional hike.

This research was also made possible by the labor of countless people whom I do not know personally. I would like to thank the contributors to Rust, glibc, GDB, rr, Valgrind, strace, Git, and the many other free and/or open-source software projects that make computers worth using. I am constantly impressed by the quality of today's libre development tooling, which greatly simplifies the implementation and debugging of even low-level code. While I have done little to contribute to the ecosystem of tools, you have my appreciation. Your work leaves me in awe of my predecessors who conducted operating systems research before such tools were available. I must also thank the anonymous reviewer who wrote the following kind words about our eponymous conference paper, words made especially inspiring by my own teaching aspirations:

> It seems like a very old-school ATC style submission, which is great. When I was teaching undergraduates about high-concurrency userspace services and the trade-offs between coroutines and event-driven programming, this would have been a nice approach to present.

I became a Ph.D. student in order to pursue a career in teaching, and thanks to the many people who placed their trust in me, I have had numerous opportunities for practice throughout my graduate studies. Charlie Garrod served as a sort of teaching mentor during the early half of my studies, something that every doctoral student who longs to teach needs to find. Bill Scherlis enabled me to take a semester-long leave of absence to serve as a sabbatical replacement for one of the computer science teachers at a local private high school, where I had (among other responsibilities) the amazing experience of designing and teaching a seminar course on computer architecture to a small group of very motivated students. Brian Railing twice invited me to serve as co-instructor of record for Carnegie Mellon's well-known 15-213 Introduction to Computer Systems course. Tom Cortina asked me to teach a very compressed six-week instance of the data structures course, 15-122 Principles of Imperative Computation, during my last summer in the program. Fellow doctoral student Kyle Liang volunteered to co-instruct with me; both of us were new to the course, and I could not have done it without him, our incredibly devoted staff of nineteen undergraduate teaching assistants, or the regular guidance of Iliano Cervesato. Later that year, when the public high school in neighboring Mt. Lebanon temporarily lost its computer science teacher, Dave was willing to continue paying for me even as I substitute taught there three days per week, just four months before my defense and before most of this dissertation was written! Special thanks go to Deb Cavlovich, the administrator of our program, for petitioning the department head for the exception to the rules that made this arrangement possible. And my longtime friend Connor Brem has my endless gratitude for subsequently *using his vacation time* to assume the role and teach the 150 high school students for the next two months while I wrote and defended this document.

Numerous others have supported my growth as a teacher in other ways. Michael Hilton taught a course on computer science pedagogy. Along with Françeska Xhakaj and Brian, he went on to found a semiweekly discussion series that now brings together teaching types from across the School of Computer Science. Dave, Guy Blelloch, and Tom funded my three successful—and one deeply unsuccessful, thanks to a global pandemic—trips to the SIGCSE computer science education conference. Erica Weng organized a series of undergraduate research mixers that directly led to a different sort of teaching experience for me, when then-freshman Yosef Alsuhaibani implemented the *strobelight* RPC system while serving as my summer research assistant.

Thank you to my entire family for always supporting my education. I attribute both my love of learning and my teaching aspirations to a tight-knit extended family that cherishes education and encourages everyone to follow their dreams rather than money. Thank you to my many aunts and uncles and older cousins for all the gatherings and outdoor adventures, and to my younger cousins for tolerating it when I force fed you computer science concepts from a young age. I would not have pursued (or successfully completed) a doctorate without my parents' encouragement, and let the record show that my father was right to insist that Carnegie Mellon's offer of a guaranteed stipend was a Big Deal. Dad, that very provision did rescue me from at least one funding shortfall, so you officially get to say, "I told you so."

Last, but certainly not least, my thanks go out to my partner Yvonne Marcoux, who selflessly took over my executive function for the first two months of this year while I locked myself in the apartment and wrote the majority of this dissertation. Without you, it would have been a miserable existence indeed, and the quality of the end product would be that much worse. Thanks for all the bicycling, camping, and road trips when I had time, and for putting up with and supporting me when I was busy and boring. Here is to many more adventures together!

*To my students,*
*past, present, and future*

# Chapter 1

# Introduction

The abstraction most fundamental to modern programs is the **function**, a section of code that expects zero or more data inputs, performs some computation, and produces zero or more outputs. It is a structured control flow primitive that obeys a strict convention: whenever invoked from one of its **call sites**, a function runs from beginning to (one possible) end, at which point execution resumes in the **caller** just after the call site. It is also a **synchronous** primitive; that is, these steps happen sequentially and in order. Because processors conceptually implement synchronous computation, scheduling a function is as trivial as instructing the processor to jump from the call site to its starting address, then later jump back to the (saved) address of whatever follows the call site. Thus, the program continues executing throughout, with no inherent need for intervention by an external scheduler or other utility software.

Note, however, that just because the program has retained control does not mean the programmer has. Precisely because functions represent an abstraction, the programmer who calls one is not necessarily familiar with its specific implementation. This can make it hard to predict the function's duration, yet calling it requires the programmer to trust it to eventually finish and relinquish control. The programmer may have made a promise (a "service-level agreement") that their whole program will complete within a specified timeframe; unfortunately, they cannot certify their compliance without breaking the abstraction and examining the internals of each function they call. Even then, untrusted or unpredictable input may make a function's performance characteristics unclear: Perhaps the function solves a problem that is known to be intractable for certain cases, but such inputs are difficult to identify *a priori*. Perhaps it performs format decoding or translation that is susceptible to attacks such as decompression bombs. Or perhaps it simply contains bugs that open it to inefficient corner cases or even an infinite loop.

Faced with such problems, the programmer is often tempted to resort to an **asynchronous** invocation strategy, whereby the function runs in the background while the programmer maintains control of the rest of the program. Common abstractions following this model include the operating system's own processes and threads, as well as the threads, coroutines, and futures (i.e., promises) provided by some libraries and language runtimes. Any use of asynchronous computation requires an external scheduler to allocate work.

Here again, the programmer is sacrificing control. Upon handing execution control to a scheduler, dependencies are no longer clear from the program's structure and must be passed to the scheduler by encoding them in synchronization constructs. (For instance, "Do not execute past this line until no one else is accessing so-and-so resource.") Sadly, it is difficult to fully communi-

cate the relevant bits of the application logic across this abstraction boundary, which can result in unintended resource-sharing effects such as priority inversion, where the scheduler chooses to run a different task than the system designer anticipated. Furthermore, each software scheduler is itself a piece of code, and because its job does not represent useful application work, any time it spends executing is pure overhead. Therefore, introducing unnecessary scheduling necessarily reduces per-processor performance.

In many cases, the *only* tool necessary to ensure timely completion of a program is preemption, the ability to externally interrupt execution. Instead of confronting this directly, current programming environments incentivize the programmer to rely on a scheduler to fix the problem, limiting them to whatever coarse timescales (often milliseconds) the OS scheduler operates at, or (in the case of userland schedulers) to cooperative scheduling that doesn't even address the problem of infinite loops. The goal of this work is to design and prototype an interface that extends the programming model with simple preemption, thereby allowing the use of functions without having to break the abstraction and examine their implementations. If a function times out, it is paused so that the programmer can later resume and/or cancel it at the appropriate time. Note that such an interface is still inherently **concurrent**; that is, the application has to manage multiple tasks at the same time. Indeed, it is now the programmer who expresses the schedule describing when to devote time to the timed code, and how much.

It bears mentioning that sometimes a system designer does need asynchronous invocation and a dedicated scheduler. Most notably, both are necessary to support **parallel** applications that actually *execute* multiple tasks simultaneously. Preemptive function calls are equally applicable in such situations because they compose with existing concurrency abstractions. In fact, we find that they make it surprisingly easy to extend existing cooperative schedulers with preemption, without adding a dependency on nonstandard OS kernel features.

## 1.1 Thesis statement

Providing language-agnostic abstractions for fine-grained preemption and function-level isolation enables the straightforward implementation of application functionality long considered prohibitively difficult, such as preemptive user threads and asynchronous task cancellation.

## 1.2 Structure and contributions

This dissertation motivates and refines a new programming abstraction for calling a function with a timeout (Chapters 2 and 5), and also introduces a separate supporting primitive for providing memory isolation within a process (Chapter 3). It presents possible applications for memory isolation (Chapter 4) and timed function calls (Chapters 7, 8, and 9). It also includes an analysis of the barriers to and a path toward achieving automatic resource cleanup upon cancellation of unfinished work (Chapter 6).

The remainder of the dissertation proceeds as follows:

**Function calls with timeouts (Chapter 2)** We examine prior approaches to running timed code from the literature, triaging the state of the art's shortcomings. In the process, we rediscover

a nigh-forgotten interface for making timed function calls, Scheme engines. The interface is elegant, but only capable of handling purely functional code. Drawing inspiration from it, we devise a novel interface for calling impure *preemptible functions*. We set the goal of language agnosticism, aiming to demonstrate support for unmanaged systems programming languages (because they provide few abstractions that might be unavailable in other settings). We also observe that using preemptible functions in an application introduces concurrency that creates unsoundness arising from shared state, specifically nonreentrancy. The application developer is unable to address the problem, so we conclude that doing so is a prerequisite for implementing preemptible functions.

**Nonreentrancy and selective relinking (Chapter 3)** We confront nonreentrancy, a program property that permeates the contemporary systems stack from the operating system up, and which poses a fundamental safety challenge to preempting impure code. To overcome this hazard, we introduce a new form of memory isolation called *selective relinking*. Crucially, this new primitive operates at a granularity finer than a kernel thread; in fact, it can be applied at the level of function calls, as is needed to support preemptible functions.

**Rethinking POSIX safety (Chapter 4)** We examine the POSIX safety concepts, the rules that govern what certain parts of a Unix program may—or may not—do. In particular, signal handlers are ordinarily only allowed to call a restricted subset of the available standard functions, and cancelable threads are conventionally barred from using operating system facilities altogether. We show how selective relinking can be applied to lift either of these restrictions and enable *safe signal handlers* or *asynchronously cancelable threads*. While our implementations are only intended as a demonstration, we find that selective relinking makes both of these seemingly Herculean tasks simple enough to make for short instructive examples.

**Function calls with timeouts, revisited (Chapter 5)** We return to the topic of preemptible functions and illustrate *how to implement them* atop the existing systems stack. In the process, we specialize what we had designed as a C interface for the more modern Rust programming language. The result is a platform that exhibits both seamless interoperability with legacy code and harmonious integration with the memory and concurrency safety features of Rust's type system. The discussion serves both as a demonstration of the kind of considerations that would go into integrating preemptible functions into other language ecosystems and as an example of a more advanced use of selective relinking.

**Resource cleanup and async unwinding (Chapter 6)** We discuss the problem of resource leaks that can occur when asynchronously cancelling code. We develop an approach and proof-of-concept system for ensuring the cleanup of cancelled code's own resources with assistance from the compiler. We do this by repurposing exception handling, which requires us to repair *asynchronous stack unwinding*. Compilers have struggled to support this feature, but we devise runtime workarounds to handle the troublesome cases, introducing only minimal additional overhead on the normal execution path.

**Preemptive userland threading (Chapter 7)** We show how preemptible functions compose with other concurrency abstractions, namely threads and futures. We create a *thread library that implements preemptive scheduling in userland* and still supports unmanaged code. To accomplish this, we construct a preemptible future type, which serves as a language-specific adapter of the preemptible futures interface. We use this to add preemption to the thread pool of an existing futures executor.

**Preemptive remote procedure calls (Chapter 8)** We observe a pain point common to RPC systems that support impure code: it is universally incumbent on the developer of each server-side function to manually, and periodically, check whether it has exceeded its service-level agreement. We describe how a first-year undergraduate student used our preemptible functions abstraction to build a *preemptive RPC system* without this limitation in two months. Its server uses a single-process architecture but isolates each request within a preemptible function. It is also capable of memoizing both fully- and partially-computed requests; in the latter case, a repeat request resumes the paused computation from wherever it left off.

**Microsecond-scale microservices (Chapter 9)** We discuss how preemptible functions could be applied to address the problem of *invocation latency in serverless computing*. Whereas contemporary systems typically place each tenant in its own separate container comprising one or more processes and a virtual filesystem, we propose consolidating numerous tenants into a single worker process. Preemptible functions would provide compute time isolation, whereas memory isolation would be achieved by requiring tenants to submit only code that could be proven safe and restricting them to a vetted set of dependencies.

**Conclusions and continuations (Chapter 10)** We summarize our work, including noteworthy technical challenges we faced and selected lessons for future systems builders. We propose possible future research directions.

“ ‘Does everyone just believe what he wants to?’
‘As long as possible. Sometimes longer.’ ”
— Isaac Asimov, *The Gods Themselves*

# Chapter 2

# Function calls with timeouts

In this chapter, we introduce the design of lightweight preemptible functions, our abstraction for making ordinary function calls with a timeout. We will cover the implementation in Chapter 5, after the intervening chapters have introduced a supporting abstraction for memory isolation and shown how to use it.

One thing that distinguishes preemptible functions is that their invocation is synchronous; that is, the program does not continue executing the code following the call until the callee has made some progress (though not necessarily run to completion). This stands in contrast to abstractions with asynchronous invocation. The thread and callback-based future abstractions are like preemptible functions in that structuring code for them involves writing a function describing the task (a thread main function or a callback, respectively). But these traditional abstractions differ in that this function does not necessarily begin executing until later: a scheduler or event loop in the runtime or operating system manages all of the program's tasks and decides when to invoke each one.

The other distinguishing feature of preemptible functions is that they are preemptive, meaning that interruption is external and can occur at (almost) any point in their execution. Abstractions such as futures and user threads are generally cooperative. As such, interruption is internal and control transfers to another task only when the executing one explicitly yields control (or calls a library function that does so on its behalf). Table 2.1 classifies concurrency abstractions based on their type of invocation and interruption.

| | | task interruption | |
|---|---|---|---|
| | | cooperative | preemptive |
| task invocation | asynchronous | user threads<br>callback-based futures | kernel threads |
| | synchronous | “async” futures<br>(`async`/`await`) | **preemptible functions** |

**Table 2.1:** Concurrency abstractions classified by type of invocation and interruption

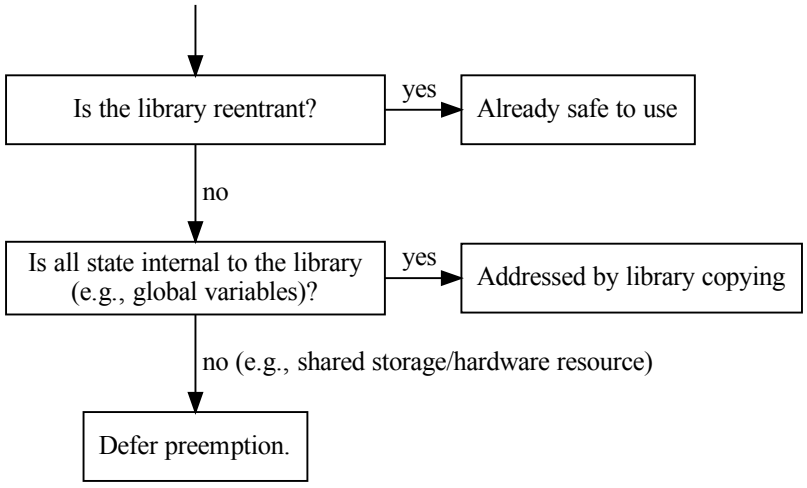



**Figure 2.1:** Taxonomy of support for library code. It is difficult to determine whether a library is fully reentrant, so in practice we always apply one of the two mitigations. Library copying is used by default, but deferred preemption is needed to preserve the semantics of `malloc()` and users of uncopyable resources such as file descriptors or network adapters.

From the table, it is apparent that the term **asynchronous** is overloaded. In the context of invocation, it is synonymous with "background." Confusingly, in the context of futures, async functions are those that can use the `await` keyword to insert a yield point that takes a *synchronous* function call off the critical path. POSIX has a third meaning for the term: In the context of signals and cancellation, it means "preemptive." Examples of this usage include the phrases "asynchronous cancellation" and "async safety."

## 2.1 Motivation

After years of struggling to gain adoption, the coroutine has finally become a mainstream abstraction for cooperatively scheduling function invocations. Languages as diverse as C#, JavaScript, Kotlin, Python, and Rust now support "async functions," each of which expresses its dependencies by "awaiting" a **future** (or promise); rather than polling, the language yields if the awaited result is not yet available.

Key to the popularity of this concurrency abstraction is the ease and seamlessness of parallelizing it. Underlying most futures runtimes is some form of green threading library that maps user threads corresponding to futures onto kernel threads serving as workers (i.e., an M:N thread pool). Without uncommon kernel support (e.g., scheduler activations [3]), however, this logical threading model renders the operating system unaware of individual tasks, meaning context switches are purely cooperative. This limitation is common among userland thread libraries, and illustrates the need for a mechanism for *preemptive* scheduling at finer granularity than the kernel thread.

In this dissertation , we propose an abstraction for calling a function with a timeout: Once

invoked, the function runs on the same thread as the caller. Should the function time out, it is preempted and its execution state is returned as a continuation in case the caller later wishes to resume it. The abstraction is exposed via a wrapper function reminiscent of a thread spawn interface such as `pthread_create()` (except *synchronous*). Despite their synchronous nature, **preemptible functions** are useful to programs that are parallel or rely on asynchronous I/O; indeed, we later demonstrate how our abstraction composes with futures and threads.

The central challenge of introducing preemption into the contemporary programming model is supporting existing code. Despite decades of improvement focused on thread safety, modern systems stacks still contain critical nonreentrancy, ranging from devices to the dynamic memory allocator's heap region. Under POSIX, code that interrupts other user code is safe only if it restricts itself to calling async-signal-safe (roughly, reentrant) functions [63]. This restriction is all too familiar to those programmers who have written signal handlers: it is what makes it notoriously difficult to write nontrivial ones. Preemption of a timed function constitutes its interruption by the rest of the program. This implies that *the rest of the program* should be restricted to calling reentrant functions; needless to say, such a rule would be impractical. Addressing this problem is one of the main contributions of this dissertation . Our main insight here, as shown in Figure 2.1, is that some libraries are naturally reentrant, while many others can be made reentrant by automatically cloning their internal state so that preempting one invocation does not leave the library "broken" for concurrent callers.

The most obvious approach to implementing preemptible functions is to map them to OS threads, where the function would run on a new thread that could be cancelled upon timeout. Unfortunately, cancelling a thread is also hard. Unix's pthreads provide asynchronous cancelability, but according to the Linux documentation, it

> is rarely useful. Since the thread could be cancelled at *any* time, it cannot safely reserve resources (e.g., allocating memory with `malloc()`), acquire mutexes, semaphores, or locks, and so on... some internal data structures (e.g., the linked list of free blocks managed by the `malloc()` family of functions) may be left in an inconsistent state if cancellation occurs in the middle of the function call [51].

The same is true on Windows, whose API documentation warns that asynchronously terminating a thread

> can result in the following problems: If the target thread owns a critical section, the critical section will not be released. If the target thread is allocating memory from the heap, the heap lock will not be released...

and goes on from there [65].

One might instead seek to implement preemptible functions via the Unix `fork()` call. Assuming a satisfactory solution to the performance penalty of this approach, one significant challenge would be providing bidirectional object visibility and ownership. In a model where each timed function executes in its own child process, not only must data allocated by the parent be accessible to the child, but the opposite must be true as well. The fact that the child may terminate before the parent raises allocation lifetime questions. And all this is without addressing the difficulty of even calling `fork()` in a multithreaded program without subsequently calling `exec()` from the newly-created child process: because doing so effectively cancels all threads in the child process

| System | Preemptive | Synchronous | Dependencies | | Third-party code support | |
|---|---|---|---|---|---|---|
| | | | In userland | Works without GC | Preemptible | Works without recompiling |
| *Scheme engines* | ✓* | ✓ | ✓ | | † | ✓ |
| *Lilt* | | ✓ | ✓ | | †* | — |
| *goroutines* | ✓ | | ✓ | | †* | — |
| *C∀* | ✓ | | ✓ | ✓ | †* | — |
| *RT library* | ✓ | | ✓ | ✓ | | ✓ |
| *Shinjuku* | ✓ | | | ✓ | † | |
| *libinger* | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

✓* = the language specification leaves the interaction with blocking system calls unclear
† = assuming the third-party library is written in a purely functional (stateless) fashion
†* = the third-party code must be written in the language without foreign dependencies
(beyond simple recompilation, this necessitates porting)

**Table 2.2:** Systems providing timed code at sub-process granularity

except the calling one, the child process can experience the same problems that plague thread cancellation [6].

These naïve designs share another shortcoming: in reducing preemptible functions to a problem of parallelism, they hurt performance by placing thread creation on the critical path. Thus, the state-of-the-art abstractions' high costs limit their composability. We observe that, when calling a function with a timeout, it is concurrency alone—and not parallelism—that is fundamental. Leveraging this key insight, we present a design that *separates interruption from asynchrony* in order to provide *preemption at granularities in the tens of microseconds*, orders of magnitude finer than contemporary OS schedulers' millisecond timescales. Our research prototype[1] is implemented entirely in userland, and requires neither custom compiler or runtime support nor managed runtime features such as garbage collection.

This dissertation makes three primary contributions: (1) It proposes function calls that return a continuation upon preemption, a novel primitive for unmanaged languages. (2) It introduces selective relinking, a compiler-agnostic approach to automatically lifting safety restrictions related to nonreentrancy. (3) It demonstrates how to support asynchronous function cancellation, a feature missing from state-of-the-art approaches to preemption, even those that operate at the coarser granularity of a kernel thread.

## 2.2 Related work

A number of past projects (Table 2.2) have sought to provide bounded-time execution of chunks of code at sub-process granularity. For the purpose of our discussion, we refer to a portion of the program whose execution should be bounded as **timed code** (a generalization of a preemptible function); exactly how such code is delineated depends on the system's interface.

Interface notwithstanding, the systems' most distinguishing characteristic is the mechanism by which they enforce execution bounds. At one end of the spectrum are **cooperative** multitasking systems where timed code voluntarily cedes the CPU to another task via a runtime check. (This is often done implicitly; a simple example is a compiler that injects a conditional branch at the beginning of any function call from timed code.) Occupying the other extreme are **preemptive** systems that externally pause timed code and transfer control to a scheduler routine (e.g.,

[1] Our system is open source; the code is available from `efficient.github.io/#lpf`.

via an interrupt service routine or signal handler, possibly within the language's VM).

The cooperative approach tends to be unable to interrupt two classes of timed code: (1) **blocking-call** code sections that cause long-running kernel traps (e.g., by making I/O system calls), thereby preventing the interruption logic from being run; and (2) **excessively-tight loops** whose body does not contain any yield points (e.g., spin locks or long-running CPU instructions). Although some cooperative systems refine their approach with mechanisms to tolerate either blocking-call code sections [25] or excessively-tight loops [71], we are not aware of any that are capable of handling both cases.

One early instance of timed code support was the *engines* feature of the Scheme 84 language [30]. Its interface was a new `engine` keyword that behaved similarly to `lambda`, but created a special "thunk" accepting as an argument the number of ticks (abstract time units) it should run for. The caller also supplied a callback function to receive the timed code's return value upon successful completion. Like the rest of the Scheme language, engines were stateless: whenever one ran out of computation time, it would return a replacement engine recording the point of interruption. Engines' implementation relied heavily on Scheme's managed runtime, with ticks corresponding to virtual machine instructions and cleanup handled by the garbage collector. Although the paper mentions timer interrupts as an alternative, it does not evaluate such an approach.

*Lilt* [71] introduced a language for writing programs with statically-enforced timing policies. Its compiler tracks the possible duration of each path through a program and inserts yield operations wherever a timeout could possibly occur. Although this approach requires assigning the execution limit at compile time, the compiler is able to handle excessively-tight loops by instrumenting backward jumps. Blocking-call functions remained a challenge, however: handling them would have required operating system support, reminiscent of *Singularity*'s static language-based isolation [21].

Some recent languages offer explicit userland threading, which can be used to support timed code. One example is the Go language's [25] *goroutines*, which originally relied on a cooperative scheduler that conditionally yielded at function call sites. This caused real-world problems for tight loops, requiring affected programmers to manually add calls to the `runtime.Gosched()` yield function [10]. To address this, the language eventually migrated to a preemptive goroutine scheduler [26].

The solutions described thus far all assume languages with a heavyweight, garbage-collected runtime. However, two recent systems seek to support timed code with fewer dependencies: the *C*∀ language [13] and a C thread library for realtime systems (here, "*RT*") developed by Mollison and Anderson [47]. Both perform preemption using timer interrupts, as proposed in the early Scheme engines literature. They install a periodic signal handler for scheduling tasks and migrating them between cores, a lightweight approach that achieves competitive scheduling latencies. However, as explained later in this section, the compromise is interoperability with existing code.

*Shinjuku* [35] is an operating system designed to perform preemption at microsecond scale. Built on the Dune framework [7], it runs tasks on a worker thread pool controlled by a single centralized dispatcher thread. The latter polices how long each task has been running and sends an inter-processor interrupt (IPI) to any worker whose task has timed out. The authors study the cost of IPIs and the overheads imposed by performing them within a VT-x virtual machine, as required by Dune. They then implement optimizations to reduce these overheads at the expense of Shinjuku's isolation from the rest of the system.

As seen in Section 2.1, nonreentrant interfaces are incompatible with externally-imposed time limits. Because such interfaces are prolific in popular dependencies, no prior work allows timed code to transparently call into native third-party libraries. Scheme engines and Lilt avoid this issue by only supporting functional code, which cannot have shared state. Go is able to preempt goroutines written in the language itself, but a goroutine that makes any foreign calls to other languages is treated as nonpreemptible by the runtime's scheduler [20]. The C∀ language's preemption model is only safe for functions guarded by its novel monitors: the authors caution that "any challenges that are not [a result of extending monitor semantics] are considered as solved problems and therefore not discussed." With its focus on realtime embedded systems, RT assumes that the timed code in its threads will avoid shared state; this assumption mostly precludes calls to third-party libraries, though the system supports the dynamic memory allocator by treating it as specifically nonpreemptible. Rather than dealing with shared state itself, Shinjuku asks application authors to annotate any code with potential concurrency concerns using a nonpreemptible `call_safe()` wrapper.

## 2.3 Preemptible functions: *libinger*

We observe that today's concurrency abstractions offer either synchronous invocation or preemptive scheduling, but not both. On one hand we have futures, which are now synchronous[2] but purely cooperative. On the other are kernel threads, which are preemptive but asynchronous. We bridge this gap by introducing a novel abstraction that provides synchrony *and* preemption for unmanaged languages.

Doing so requires confronting the nonreentrancy problems that have long doomed attempts to support asynchronous cancellation outside of purely functional contexts. This turns out to be a slightly harder problem than safely supporting concurrency, so in addition to cancellation, we get the ability to externally pause for free.

To address the literature's shortcomings, we have developed *libinger*,[3] a library providing a small API for timed function dispatch (Listing 2.1):

- `launch()` invokes an ordinary function `func` with a time cap of `time_us`. The call to `launch()` returns when `func` completes, or after approximately `time_us` microseconds if `func` has not returned by then. In the latter case, *libinger* returns an opaque continuation object recording the execution state.

- `resume()` causes a preemptible function to continue after a timeout. If execution again times out, `resume()` updates its continuation so the process may be repeated. Resuming a function that has already returned has no effect.

Listing 2.2 shows an example use of *libinger* in a task queue manager designed to prevent latency-critical tasks from blocking behind longer-running ones. The caller invokes a task with a timeout. If the task does not complete within the allotted time, the caller saves its continuation in the task queue, handles other tasks, and later resumes the first task.

[2]Code interfacing with futures via the "async/await" continuation passing style is now prolific.

[3]In the style of GNU's *libiberty*, we named our system for the command-line switch used to link against it. As the proverb goes, "Don't want your function calls to linger? Link with `-linger`."

```
struct linger_t {
        bool is_complete;
        cont_t continuation;
};

linger_t launch(Function func, u64 time_us, void *args);
void resume(linger_t *cont, u64 time_us);
```

**Listing 2.1:** Preemptible functions core interface

```
linger = launch(task, TIMEOUT, NULL);
if (!linger.is_complete) {
        // Save `linger` to a task queue to resume later
        task_queue.push(linger);
}

// Handle other tasks
...

// Resume `task` at some later point
linger = task_queue.pop();
resume(&linger, TIMEOUT);
```

**Listing 2.2:** Preemptible function usage example

### 2.3.1 Design principles

Although we are introducing a new concurrency abstraction, we have striven to keep the interface simple and understandable. The following design principles have guided this effort:

- **We do not assume that users need asynchrony.** Hence, preemptible functions *run on the same kernel thread as their caller*. This is good for performance (especially invocation latency), but it is also important to be aware of; for instance, it means that a preemptible function will deadlock if it attempts to acquire a lock held by its caller, or vice versa. Of course, some users may need asynchrony. The preemptible function abstraction composes naturally with both threads and futures (Chapter 7), so there is no need to reinvent the wheel.

- **We assume that simply calling a preemptible function is the common use case.** As such, the `launch()` wrapper both constructs and invokes the preemptible function rather than asking the user to first employ a separate constructor. Users wishing to separate the construction and invocation operations can pass the sentinel `0` as the timeout, then later use `resume()` to start execution.

- **We favor a simple, language-agnostic interface.** The fact that our interface centers on a higher-order function in the style of the `pthread_create()` and `spawn()` wrapper functions means that using preemptible functions looks similar regardless of the programming language. Currently, *libinger* provides bindings for both C and Rust. If and when we add bindings for other languages, we expect them to have the same feel; in the meantime, other languages can use preemptible functions (unsafely) through their C foreign-function interfaces. We considered adhering to the futures interface instead, but decided against it because each language has its own incompatible variant thereof. The relative ease of building a futures adapter type (Chapter 7) affirms our decision.

- **We keep argument and return value passing simple yet extensible.** Because Rust supports closures, the Rust version of `launch()` accepts only nullary functions: those seeking to pass arguments should just capture them from the environment. C supports neither closures nor generics, so the C version of `launch()` accepts a single `void *` argument that can serve as an in/out parameter. It occupies the last position in the parameter list to permit (possible) eventual support for variable argument lists.

- **We choose defaults to favor flexibility and performance.** When a preemptible function times out, *libinger* assumes the caller might later want to resume it from where it left off. As such, both `launch()` and `resume()` pause in this situation; this incurs some memory and time overhead to provide a separate execution stack and package the continuation object, but exhibits lower latency than asynchronous cancellation. If the program does require cancellation, we provide ways to explicitly request it (Chapter 5).

- **In addition to preemption, we offer the option to yield.** This feature enables the construction of higher-level synchronization constructs tailored to preemptible functions (Chapter 5). It also allows preemptible functions to coexist with cooperatively-scheduled tasks such as futures (Chapter 7).

## 2.4 The preemptible functions ecosystem

In divorcing preemption from asynchronous invocation, preemptible functions disentangle interruption from parallelism. Indeed, *libinger* does not provide a task scheduler because the only decision it makes is whether to pause the currently executing code. Whenever it opts to do so, it unconditionally returns control to the preemptible function's caller.

This design allows client code to pick and choose the level of runtime support it needs. If it only invokes preemptible functions synchronously and makes all scheduling decisions itself, it can link directly against *libinger* and use the interface presented in Section 2.3. If it prefers to delegate scheduling to a runtime, we also provide *libturquoise*, a preemptive futures executor offering an event loop and thread pool (Chapter 7).

Figure 2.2 shows the dependency relationship between *libinger* and *libturquoise*, in the context of the other software components developed for this dissertation. Notably, both libraries support nonreentrancy by depending on another library called *libgotcha*, which provides a novel abstraction of its own for enforcing isolation boundaries. The *libinger* library is implemented in approximately 2,500 lines of Rust; *libgotcha* comprises another 3,000 lines of C, Rust, and x86-64 assembly.

### 2.4.1 Automatic handling of shared state: *libgotcha*

As we found in Section 2.1, a key design challenge facing *libinger* is the shared state problem: Suppose a preemptible function $F$ calls a stateful routine in a third-party library $L$, and that $F$ times out and is preempted by *libinger*. Later, the user invokes another timed function $F_0$, which also calls a stateful routine in $L$. This pattern involves an unsynchronized concurrent access to $L$. To avoid introducing such bugs, *libinger* must hide state modifications in $L$ by $F$ from the execution of $F_0$.

One non-solution to this problem is to follow the approach taken by POSIX signal handlers and specify that preemptible functions may not call third-party code, but doing so would severely limit their usefulness (Section 2.2). We opt instead to automatically and dynamically create copies of $L$ to isolate state from different timed functions. Making this approach work on top of existing systems software required solving many design and implementation challenges, which we cover when we introduce *libgotcha* in Chapter 3.

Note that preemptible functions are still a concurrency abstraction, and our automatic handling of shared state internal to dependencies does not exempt the author of a preemptible function from writing safe concurrent code.

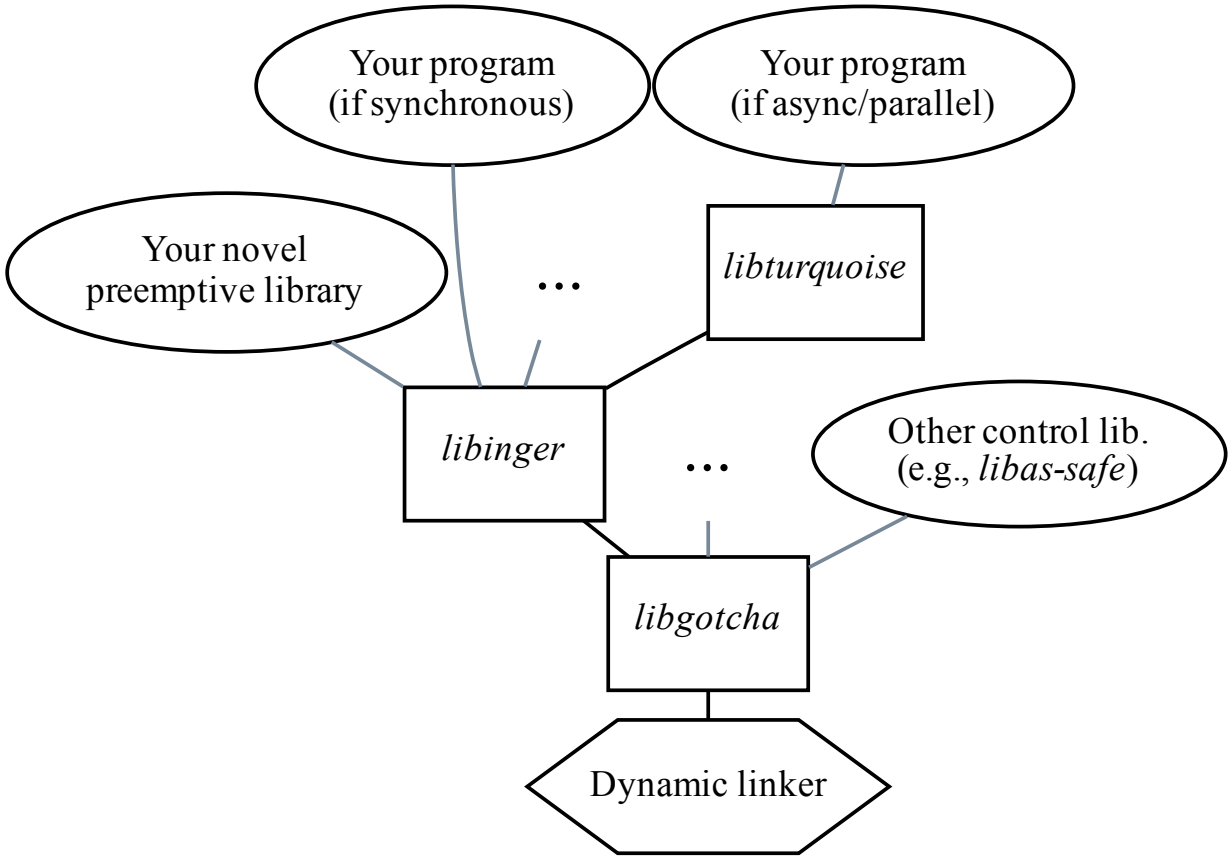



**Figure 2.2:** Preemptible functions software stack. Hexagonal boxes show the required runtime environment. Rectangular boxes represent components implementing the preemptible functions abstraction. Ovals represent components built on top of these. A preemptible function's body (i.e., `func`) may be defined directly in your program, or in some other loaded library.

> “ ‘Alien superweapons were used,’ Alex said, walking into the room,
> sleep-sweaty hair standing out from his skull in every direction.
> ‘The laws of physics were altered, mistakes were made.’ ”
> — James S. A. Corey, *Nemesis Games*

# Chapter 3

# Nonreentrancy and selective relinking: the *libgotcha* runtime

In Section 2.4.1, we saw that it is not safe in general for a preemptible function to call into stateful code that was written without the preemptible function abstraction in mind. However, such code is prolific in the modern systems stack, and in order to support interoperability with it, we need to automatically transform the program to fix the safety hole. This chapter introduces a novel abstraction for memory isolation and constructs a software system that implements it, dubbed *libgotcha*; then, we conclude the chapter by presenting performance metrics.

Despite the name, it is more like a runtime that isolates hidden shared state within an application. Although the rest of the program does not interact directly with *libgotcha*, its presence has a global effect: once loaded into the process image, it employs a technique we call **selective relinking** to dynamically intercept and reroute many of the program's function calls and global variable accesses.

The goal of *libgotcha* is to establish around every preemptible function a memory isolation boundary encompassing whatever third-party libraries that function interacts with (Section 2.4.1). The result is that the only state shared across the boundary is that explicitly passed via arguments, return value, or closure—the same state the application programmer is responsible for protecting from concurrency violations (Section 2.4). Listing 3.1 shows the impact on an example program, and Figure 2.1 classifies libraries by how *libgotcha* supports them.

Note that *libgotcha* operates at runtime; this constrains its visibility into the program, and therefore the granularity of its operation, to shared libraries. It therefore assumes that the programmer will dynamically link all third-party libraries, since otherwise there is no way to tell them apart from the rest of the program at runtime. We feel this restriction is reasonable because a programmer wishing to use *libinger* or *libgotcha* must already have control over their project's build in order to add the dependency.

Before introducing the *libgotcha* API and explaining selective relinking, we now briefly motivate the need for *libgotcha* by demonstrating how existing system interfaces fail to provide the required type of isolation. Our discussion in this chapter uses *libinger* as a motivating example of a *libgotcha* user, as this configuration was the inspiration for the runtime's creation. However, we have found the described techniques to be general and equally relevant to applications beyond timed functions. As such, *libgotcha* exposes a general API that allows any **control library** to configure its behavior for the process. We give more details later in the chapter, and study other examples of control libraries in Chapter 4.

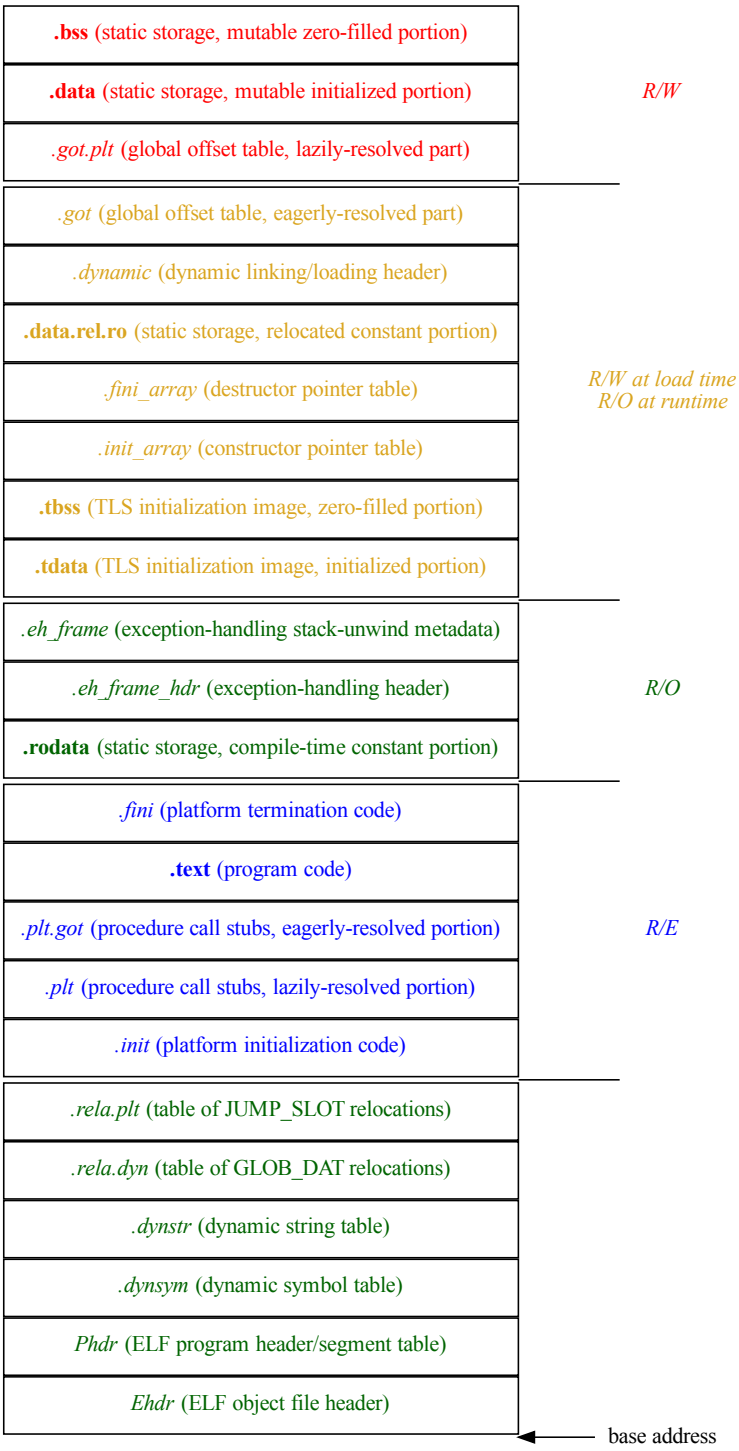



**Figure 3.1:** Layout of a typical module within the process image. **Bold** sections contain program data; *italicized* ones contain metadata for the runtime.

```
static bool two;
bool three;

linger_t caller(const char *s, u64 timeout) {
        stdout = NULL;
        two = true;
        three = true;
        return launch(timed, timeout, s);
}

void timed(void *s) {
        assert(stdout); // (1)
        assert(two); // (2)
        assert(three); // (3)
}
```

**Listing 3.1:** Demo of isolated (1) vs. shared (2&3) state

## 3.1 A brief tour of linking

We begin with background about linking, a two-stage process that ultimately produces an in-memory **process image** containing a program's code, all the data it needs to execute, and the code and data of all its dependencies. Linking operates on **object files** that can take the form of either an **executable** or a **shared library**. Once a program is running, its process image contains a region corresponding to each loaded object file. We will refer to each such region as a **module**, regardless of whether it corresponds to an executable or a shared library. Each module is divided into logical **sections**, each containing a particular type of information. Figure 3.1 shows a typical module's layout; notice that it contains both data corresponding to the source code and generated metadata for runtime consumption.

The linking process occurs in two parts. Static linking occurs at compile time and forms the last step of the traditional build process. Dynamic linking occurs at a phase of runtime known as **load time**, which starts before the program has been loaded from disk or the language runtime initialized.

### 3.1.1 Static linking

Invoking the `cc` compiler driver does more than just compile C code: it runs the C preprocessor `cpp`, the C compiler (`cc1` in GCC's case), then the static linker `ld`.

The output of the second step is a relocatable object file containing code and data with referenced addresses identified by named **symbols**. In a relocatable object file, symbol **references** such as instructions making function calls or accessing global variables are encoded with a null address as a placeholder. Each object file contains a **relocation table** in a separate section that associates each placeholder with a symbol name, which may or may not be located in the same file. Each object file also contains a **symbol table** to identify the symbols it defines and associate

them with the file offset of their definition. Note that only non-`static` C symbols generate global symbol table entries that can be referenced from other object files; this keyword is confusingly named and does not refer to static linking. The compiler's ultimate output is one relocatable object file for each source file.

The static linker is responsible for combining one or more relocatable object files into a single executable or shared library, where either type of output file is ready for loading into memory for execution. This process consists of verifying that there is a definition corresponding to each symbol reference, unifying the sections across object files and choosing a final address (or relative address) for each symbol, encoding the chosen addresses at the location recorded in each relocation table entry, and writing the resulting file to disk. This output file does not preserve the relocation table because the linker has already fixed the null pointers it described. The file does contain a symbol table because it can be useful for debugging (e.g., to generate stack traces), but this can be removed using the `strip` utility without affecting the program semantics.

With the exception of macOS, most modern Unix systems use ELF (Executable and Linkable Format) object files. One advantage of this format is that executables and shared libraries are themselves ELF object files.

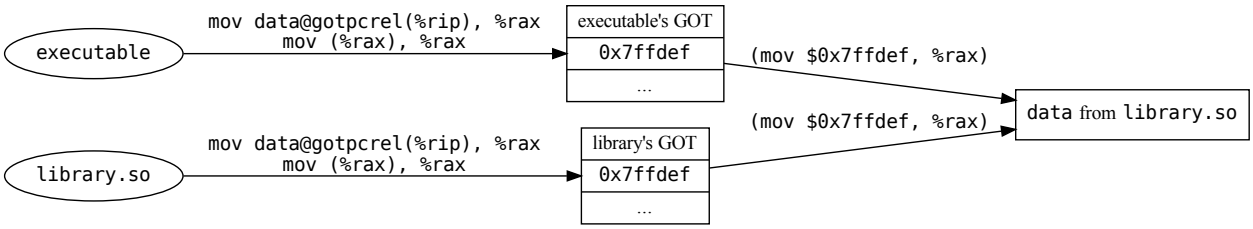
### 3.1.2 Dynamic linking

Static linking allows programs to reuse "libraries" of precompiled object files, but each program must be built with its own copy of all its libraries within the executable. This means that every time an application is loaded, its libraries' code and data must be read back from disk, even if another running program uses the same libraries; it also means that updating a library requires recompiling all dependent programs installed on the system. Dynamic linking solves both problems by separating libraries into separate files that are not read until the executable runs.[1]

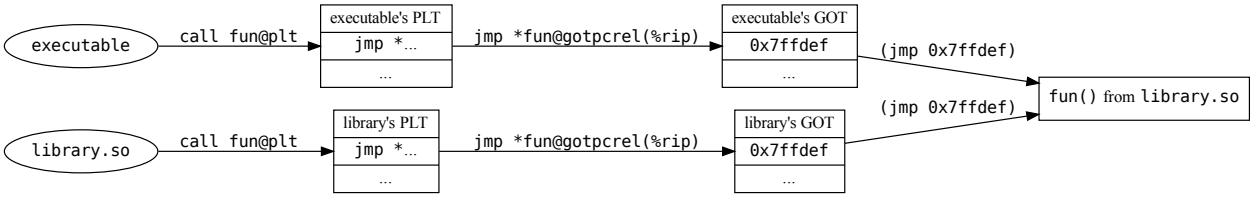
By splitting libraries into their own files, dynamic linking introduces a build-time challenge: the relative position and offset of modules cannot be known until runtime. As such, rather than performing the relocations for inter-module symbol references, the static linker leaves the placeholder addresses and adds a separate dynamic relocation table and dynamic symbol table into the output object file. Unlike the tables used for static linking, these are needed to launch the program, so tools such as `strip` leave them in place. For executables, the linker also writes the path to an "interpreter" program into the ELF program header.

When asked to load a program that declares an interpreter, the kernel loads and jumps to the interpreter instead of the executed program. Usually, this interpreter is the system **dynamic linker**, traditionally named `ld.so`. Before jumping into the program code, the dynamic linker loads all the modules and processes the entries in each of their dynamic relocation tables. The relocations are not restricted to modifying writeable memory: they can update constant global data and even executable instructions. Even if they leave the code unchanged, its position relative to the rest of the module matters. These points are critical to our use case, as they mean that in order to duplicate modules' data, we must also duplicate their code.

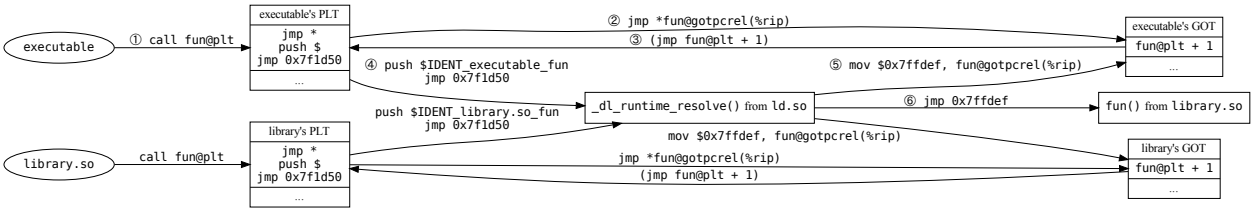
[1] Specifically, this separation obviates the need to read the files from disk multiple times because the runtime maps them into the process image using the `mmap()` family of system calls. The kernel tracks regions that are already mapped and serves recurring requests from memory instead of disk, mapping to the same physical memory if the pages are read-only or creating copy-on-write page mappings otherwise.




**(a)** Reading a library's global variable: `size_t tmp = data;`




**(b)** Calling an eagerly-resolved library function: `fun()`




**(c)** Calling a lazily-resolved library function. In step ⑤, the dynamic linker memoizes the resolved address into the GOT; subsequent calls proceed as above.

**Figure 3.2:** Table references required to reference global symbols in dynamically-linked programs

Another consequence of relocations being able to alter read-only memory is that the dynamic linker must change the page protections of these regions after it has finished processing relocations. To support this, the compiler splits up module components into fine-grained sections by purpose. Non-`const` global variables are placed in the `.data` and `.bss` sections, which must remain writeable at runtime and therefore require no special action. In contrast, `const` globals are split between the `.rodata` and `.data.rel.ro` sections based on whether they require relocation; in the latter case, the dynamic linker marks the pages read-only before passing control to the program.

If relocations routinely modified scattered locations throughout the executable `.text` section, the dynamic linker would have to change page protections on most or all of each module's code pages. This would require a lot of system calls, but it would also require copy-on-write code mappings, preventing instruction cache hits between processes using the same library. To avoid these problems, the compiler indirects references to dynamic symbols via a structure called the GOT (Global Offset Table).

The GOT is a table of relocated pointers to symbol definitions, whether those definitions are within the same module or in a different one. To avoid generating code pages that require relocations, the compiler compiles each reference to or dereference of non-`static` global data into a position-independent load of the corresponding pointer from the GOT. Figure 3.2a shows an example of the two instructions and one table reference needed for a dereference. A reference

would generate only the first `mov` instruction, as would taking a pointer to a function.

Calling a global function works differently and relies on another indirection structure called the PLT (Procedure Linkage Table), which contains code instead of pointers. For each call to a non-`static` function, the compiler generates a position-independent call to a PLT entry corresponding to the function being called. It generates a PLT entry, which is a short sequence of instructions that loads the pointer to the real definition from the GOT, then executes an indirect jump to that location. Figure 3.2b shows an example function call. As with GOT entries, there are PLT entries for functions defined both within and outside the referencing module.

Not all function calls are this simple. To save the dynamic linker some work at load time, many function calls resolve lazily on their first execution. Such resolution involves a series of jumps designed to memoize the address so that subsequent calls to the function from the same module do not repeat the expensive lookup. Figure 3.2c shows the effect of the first call to such a function:[2] ① The program calls the PLT stub, just as it would for an eagerly-resolved function. ② The PLT stub is longer (three instructions instead of one), but still begins with an indirect jump to the pointer found in the corresponding GOT entry. ③ The GOT entry initially contains the address of the PLT stub's second instruction, so the indirect jump is a no-op and merely advances the instruction pointer. ④ The rest of the PLT stub pushes a constant identifying the module and symbol onto the stack, then jumps to a symbol-lookup function in the dynamic linker. ⑤ After looking up the address of the symbol's definition, the dynamic linker uses the identifier from the stack to find and update the GOT entry in the calling module. ⑥ The dynamic linker jumps to the symbol in the defining module. Because the GOT entry has been updated, future calls proceed exactly like eagerly-resolved ones and jump directly to the symbol definition from the first instruction of the PLT stub. Of course, the GOT entries associated with lazy PLT stubs must be writeable at runtime; this is why Figure 3.1 shows the GOT as split between two sections.

The dynamic linker performs all relocations and other standard module setup automatically at load time, but the initialization process is pluggable. In particular, modules can include **constructor** functions to be invoked before control is transferred to the runtime and ultimately the program's main function. As we will see, our work leverages this feature to override certain relocations at the conclusion of load time.

## 3.2 Library copying: namespaces

Expanding a preemptible function's isolation boundary to include libraries requires providing it with private copies of those libraries. POSIX has long provided a `dlopen()` interface to the dynamic linker for loading shared objects at runtime; however, opening an already-loaded library just increments a reference count, and this function is therefore of no use for making copies. Fortunately, the GNU dynamic linker (`ld-linux.so`) also supports Solaris-style **namespaces**, or isolated sets of loaded libraries. For each namespace, `ld-linux.so` maintains a separate set of loaded libraries whose dependency graph and reference counts are tracked independently from the rest of the program [15].

[2] This representation is slightly simplified for brevity. In practice, it is undesirable to hardcode the address of a dynamic linker function into each module. Therefore, instead of jumping directly to the symbol resolver, the slow lookup path jumps to a dedicated PLT stub that loads its address from another GOT entry. Technically, there are separate identifiers for the symbol and the module, each pushed to the stack by one of these two involved PLT stubs.

```
typedef long libset_t;

bool libset_thread_set_next(libset_t);
libset_t libset_thread_get_next(void);
bool libset_reinit(libset_t);
```

**Listing 3.2:** *libgotcha* C interface

It may seem like namespaces provide the isolation we need: whenever we `launch(`*F*`)`, we can initialize a namespace with a copy of the whole application and transfer control into that namespace's copy of *F*, rather than the original. The problem with this approach is that it breaks the lexical scoping of static variables. For example, Listing 3.1 would fail assertion (2).

## 3.3 Library copying: libsets

We just saw that namespaces provide too much isolation for our needs: because of their completely independent dependency graphs, they never encounter any state from another namespace, even according to normal scoping rules. However, we can use namespaces to build the abstraction we need, which we term a **libset**. A libset is like a namespace, except that the program can decide whether symbols referenced within a libset resolve to the same libset or a different one. Control libraries such as *libinger* configure such **libset switches** via *libgotcha*'s private control API, shown in Listing 3.2.

This abstraction serves our needs: when a `launch(`*F*`)` happens, *libinger* assigns an available `libset_t` exclusively to that preemptible function. Just before calling *F*, it informs *libgotcha* by calling `libset_thread_set_next()` to set the thread's **next libset**: any dynamic symbols used by the preemptible function will resolve to this libset. The thread's **current libset** remains unchanged, however, so the preemptible function itself executes from the same libset as its caller and the two share access to the same global variables.

One scoping issue remains, though. Because dynamic symbols can resolve back to a definition in the same executable or shared object that used them, Listing 3.1 would fail assertion (3) under the described rules. We want global variables defined in *F*'s object file to have the same scoping semantics regardless of whether they are declared `static`, so *libgotcha* only performs a namespace switch when the use of a dynamic symbol occurs in a different executable or shared library than that symbol's definition.

Thus, selective relinking is selective in two ways, only affecting execution when the next libset differs from the current libset and the program references a dynamic symbol defined in a module that is not currently executing on that thread.

### 3.3.1 Detecting cross-module symbol references

Identifying which GOT entries correspond to cross-module symbol references is a multi-step process: First, we traverse the relocation table for each loaded module, cross referencing each of its relocation entries against the local module's symbol table. If the symbol table does not contain

a definition matching the relocation entry's target, we conclude that the relocation must correspond to a cross-module call. Otherwise, we check the address in the GOT entry corresponding to the relocation: If this address is outside the memory bounds of the current module, it is a cross-module call. Otherwise, if this address matches the one from the symbol table entry, it is not a cross-module call, and should be skipped. The trickiest case is when the GOT entry does not match but does point somewhere within the current module, since this means it probably still refers to the PLT stub (because the symbol reference is lazy and has not yet been resolved, as covered at the end of Section 3.1.2). In this case, we resolve the symbol early, update the GOT entry, and recheck whether it resolved to the local definition to determine whether it is a cross-module call.

## 3.4 Managing libsets

At program start, *libgotcha* initializes a pool of libsets, each with a full complement of the program's loaded object files. Throughout the program's run *libinger* tracks the libset assigned to each preemptible function that has started running but not yet reached successful completion. When a preemptible function completes, *libinger* assumes it has not corrupted its libset and returns it to the pool of available ones.

If a preemptible function is cancelled rather than being allowed to return, execution might be interrupted within a call to a library function. For this reason, *libgotcha* must treat the libset's shared state as corrupted; it provides the `libset_reinit()` function shown in Listing 3.2 to allow control libraries to inform it of such a situation so it can **reinitialize** the libset before returning it to the pool.

Our early approach to reinitialization was to unload and reload all objects in the libset by calling `dlclose()` followed by `dlmopen()`. While this approach theoretically allowed us to delegate the work to the dynamic linker, in practice it introduced significant complications.[3] Worse, it required the dynamic linker to reprocess all relocations throughout the libset, which introduced prohibitive runtime latency. We measured reinitialization taking almost 5 ms (over 10 million cycles on modern processors) on even minimal example programs [8]. With such delays, the only reasonable way for the control library to handle cancellation was to delegate the reinitialization to a separate thread to take it off the critical path; of course, this approach only works as long as the number of libsets is not a bottleneck.

We have since redesigned reinitialization around a significantly faster approach: checkpointing only portions of each module. The key insight is that, as we saw in Section 3.1, only some sections are writeable at runtime. We can therefore assume that these are the only memory regions of each module that can change. After populating the libset pool at application start, *libgotcha* iterates through each module of each libset and makes a backup copy of all its writeable regions. When a control library calls `libset_reinit()`, *libgotcha* restores each such region

[3]Most notably, some shared libraries are marked with a special configuration flag, `DF_1_NODELETE`, which prevents the dynamic linker from ever removing them once they have been loaded. Because almost all libraries depend on libc, the presence of even one such library would prevent us from reinitializing a libset. The flag is mostly used on libraries that need to monkey-patch some other loaded library, such that the two subsequently have a circular dependency. Fortunately, this was not generally a problem for us because when we unload one library from a libset, we then unload the rest. Whenever we encountered a `NODELETE` object file, we would make a special copy with the flag cleared, for loading into every namespace except the main one.

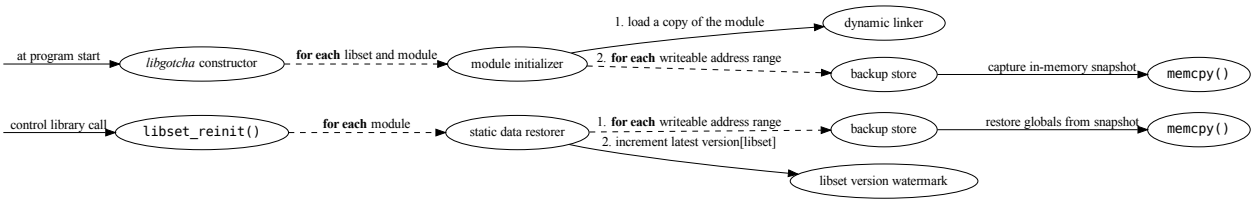



**Figure 3.3:** Libset reinitialization to support asynchronous cancellation

from the backup before returning the affected libset to the pool. We summarize this approach, which reduces the latency of reinitialization by two orders of magnitude, in Figure 3.3.[4] To avoid having to repeat relocations and rerun module constructors, we capture the backup after dynamic relocation is complete and all constructors have run; the tradeoff is that we must actually copy memory, rather than leveraging copy on write to later restore to the version on disk.

## 3.5 Selective relinking

Most of the complexity of *libgotcha* lies in the implementation of selective relinking, the mechanism underlying libset switches. To establish the libset abstraction, it must arrange to conditionally intercept cross-module symbol uses based on the currently-configured next libset.

As we saw in Section 3.1.2, whenever a program references a dynamic symbol, it looks up the address of the definition in a data structure called the global offset table (GOT). Selective relinking works by shadowing the GOT.[5] Just after the dynamic linker populates the GOTs, *libgotcha* replaces every entry that should sometimes trigger a libset switch with a fake address. It stores the original address in its shadow GOT, which is organized by the libset that houses the definition. The fake address used depends upon the type of symbol:

### 3.5.1 Intercepting function calls

When setting up selective relinking, we do not know whether a particular function call needs to be rerouted until runtime. However, the fact that dynamic function calls consult the GOT to determine which code to execute makes it efficient and relatively straightforward to receive a notification whenever they occur. To set this up, at load time, we replace each such cross-module GOT entry with the address of the special *libgotcha* function `procedure_linkage_override()`. Whenever the program tries to call one of the affected functions, control transfers to this function instead; it then checks the thread's next libset, looks up the appropriate symbol definition in the shadow GOT, and jumps to that location. Because `procedure_linkage_override()` runs between the caller's `call` instruction and the real function, it is written in assembly to avoid clobbering registers (e.g., those used for argument passing).

There is one major complication that necessitates an additional level of indirection beyond what we have described. Recall from the eagerly-resolved function calling sequence in Figure 3.2b that each function call site calls a one-instruction PLT stub that performs an indirect jump to the real definition via the GOT. This means that if we simply replaced all the GOT entries with

[4] We will address the version watermark alluded to therein later in this chapter.

[5] Hence the name *lib**got**cha*.

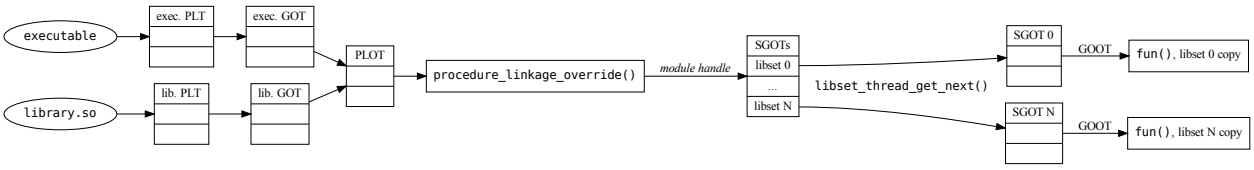



**Figure 3.4:** Calling an eagerly-resolved library function under selective relinking

the address of `procedure_linkage_override()`, that function would not know which GOT entry it was being called via, and therefore which symbol to look up. Instead, we introduce our own table of executable stubs called the PLOT (Procedure Linkage Override Table). Unlike PLT entries, ours always push an identifier indicating which function is being called. For this, we use indices into a custom data structure called the GOOT (Global Offset Override Table), which stores enough information to find the symbol's shadow GOT entries while being practical to traverse in handwritten assembly code. Figure 3.4 summarizes the modified dynamic function call sequence under selective relinking.

A subtle but important point about dynamic linking is that pointers to the same definition must compare equal, regardless of where they are obtained. The reader might notice that invocation is not the only thing a program can do with a function: it might also pass around the function's address. In fact, after taking the address, it could pass it to code within a different module, which might need to know whether a third module passed a pointer to the same function. To support such comparisons, the compiler exclusively generates eagerly-resolved relocations for any function that a particular module obtains a pointer to. This way, the `mov` to retrieve the pointer always finds the resolved address of the real definition in the GOT. To avoid breaking pointer comparison, *libgotcha* allocates a single PLOT entry for each function definition, rather than using a separate one for each call site or dependent module. This provides correct comparison semantics because all references to a particular function receive the same pointer to the shared PLOT entry, and although this pointer technically refers to the corresponding symbol's definitions in all libsets, at any one time all calls to it will only resolve to the definition in the next libset. Since at any given time there can only be one next libset, calls to pointers that compare equal always refer to the same copy of the definition.

The setup described so far works for eagerly-resolved function calls. However, recall that some function calls resolve to their definition lazily at runtime. For such calls, the dynamic linker memoizes the resolved address by updating the GOT entry, as shown in Figure 3.2c. Unfortunately, replacing this GOT entry would overwrite the PLOT pointer installed by *libgotcha* at load time, thereby preventing it from intercepting future calls to the function. The write to the GOT happens within the symbol-lookup code in the dynamic linker, so there is no way to skip it. Luckily, the dynamic linker keeps each module's dynamic relocation table in memory and uses that to determine which GOT entry to update. The *libgotcha* constructor exploits this by marking the relocation table pages writeable, changing the relocation entries corresponding to cross-module calls, then restoring the protection bits. This fools the dynamic linker's lazy symbol lookup into later updating the shadow GOT entry instead. Not only does this avoid breaking selective relinking, it also preserves memoization.

The GNU system supports a second, nonstandard type of lazy function call resolution known as indirect function calls. Compared to the lazy resolution specified in the ELF standard, this vari-

ant allows the defining library to provide custom code to resolve the address of the function definition. We must support this mechanism because glibc relies on it to select among architecture-specific implementations for reasons of optimization (e.g., `strstr()`, which leverages vector extensions on processors that have them) or simply portability (e.g., `clock_gettime()`, which depends on the best available time source). Indirect symbols require special handling because their definitions are misleading: the symbol table entry in the defining module describes the location not of the definition but of a function that returns that definition's location. If we placed such addresses into our shadow GOTs, each caller would unwittingly invoke the resolver function instead of the function it was trying to call. Whenever we see a symbol table entry marked with the `STT_GNU_IFUNC` type, we instead invoke it and place the returned pointer in the shadow GOT. A final complication is that some indirect resolver functions retrieve their return values from the GOT, so if we call them after we have manipulated it, they can create inefficient chains of calls through multiple shadowed GOT entries, or even infinite recursion where a resolver returns the PLOT stub that leads to the very shadow GOT entry it was intended for. To avoid these problems, we call all of a module's indirect resolvers eagerly and save the results into the shadow GOTs before making any changes to the real GOT.

### 3.5.2 Intercepting global variable accesses

Unlike function calls, global variable accesses do not provide an opportunity for hijacking the flow of control to detect the dereference, so we rely on operating system assistance to implement a mechanism similar to demand paging. At load time, we replace each cross-module global variable GOT entry with a carefully-chosen address within a mapped but inaccessible memory region. The program is therefore able to retrieve a "pointer to" the global variable, but whenever it attempts to read from or write to the location, it generates a segmentation fault; *libgotcha* registers a signal handler so such signals notify it rather than crashing the program. The handler disassembles the faulting instruction to determine the base address register of its address calculation and attempts to reconstruct a GOOT pointer based on the invalid address in that register. If successful, it checks the thread's next libset, retrieves the address of that libset's definition of the symbol from the appropriate shadow GOT, and replaces the base address register's contents with this address. It then returns, causing the processor to reexecute the faulting instruction with the valid address this time. From the application's perspective, it is as if dereferencing the phony pointer causes it to change into a real address. To avoid breaking applications with their own segmentation fault handlers, *libgotcha* intercepts calls to `sigaction()` and keeps a pointer to the third-party handler. Whenever its handler is unable to resolve a segmentation fault, it forwards the signal to the third-party one.

The above approach assumes that the program will read or write from the global variable, but the global variable could instead contain a function pointer. If it does and the program tries to invoke it, it will use an indirect call instruction and the jump will succeed but transfer control to an invalid location. In this situation, it is important that we do not attempt to disassemble the faulting instruction, as the instruction pointer is pointing to unreadable memory. However, we can recognize this case because the address of the fault matches the instruction pointer itself (instead of whatever address the instruction would have been trying to load data from). To find the indirect call instruction, we load the return address from the stack; this gives us the subsequent instruction, so we subtract the length of an indirect call instruction. We alter the signal

handler's context so that the operating system will transfer control back to this instruction when our handler returns, then we disassemble the instruction and follow our usual approach on its register operand.

It is possible to generate other code sequences that are incompatible with the approach (e.g., because they perform in-place pointer arithmetic rather than using a displacement-mode address calculation with a base address), so we also employ a few heuristics that consider the context of the instruction and fault. If the code is reading or writing to a faulting location that we cannot translate into a GOOT pointer, it might have applied a linear offset to the last address we successfully translated. If any of the following indicators point to this, we compute the offset between the faulting address and the last invalid address we replaced, then compute the replacement register value by adding this offset to the last replacement address we substituted:

- The client code is using the same base address register as it was when we last intercepted a global variable access. This might indicate that said code is using the register as an address accumulator, but doing so in concert with some other temporary register: because of this indirection, overwriting the register with the temporary after we had preformed the original address resolution would have left us unable to process any subsequent values accumulated into the register.

- The base address register is different than the one updated on our last interception, but the value of the latter has remained unchanged since we updated it. Because it contains a memory address we had to resolve, this strongly suggests that the client code has only executed a few instructions since then, which we can infer even if that set included one or more branch instructions.

- The current return address points to the instruction immediately following the one that last faulted, and the current base address register's value has remained the same since the faulting instruction was executed. This implies that said instruction was an indirect procedure call, and that the register was probably just used to pass a pointer argument. Because we didn't resolve the address of the indirect call until the client code was already transferring control, there was no way for it to have passed a pointer without performing arithmetic directly on the dummy address present before the call.

In our experience, these heuristics cover the common cases in compiler-generated code. We do not allow heuristics to chain (that is, we never use a heuristically-calculated address as the basis of the offset calculation for another), but we do apply heuristics multiple times based on the same base address.

While our current approach has proven successful, it does have some downsides. It is complex, relies on heuristics, and incurs a performance cost on the order of microseconds. It also suffers from a design flaw affecting large structures: it does not account for the size of globals when "allocating" them fake addresses within the inaccessible memory region. If a fake address gets assigned to a global whose size is less than the difference between that address and the end of the inaccessible region, it is possible for a correct program to dereference outside the inaccessible region and exhibit emergent undefined behavior. The ELF dynamic symbol table includes objects' sizes, so it would be possible to account for this when assigning fake addresses.

One could go further with this idea and redesign global variable interception without the need for heuristics or the dependency on a disassembler. By allocating each global its own inaccessible region matching its real size, the address of the faulting access could be made to reveal the access's offset within the global. Mapping the fake address back to a symbol would require a lookup data structure; one option is a hash table with an entry for each of our inaccessible pages. To translate a fake address, one would zero out its page offset bits, consult the lookup structure to find our metadata about the page, and use this to convert to a real symbol definition and offset within it. This final conversion could be done either by starting each global's fake region on a page boundary (at the cost of more virtual memory) or binary searching a list of the addresses within the inaccessible page that corresponded to the starts of new symbols[6] (at the cost of logarithmic worst-case lookups instead of constant average-case ones).

## 3.6 Uninterruptible functions

Unlike prior work, we support safe asynchronous pausing and cancellation on almost any instruction boundary, including within most third-party libraries without the need for configuration or code annotation. However, there are still some cases where we must briefly defer preemption.

The most obvious is the `malloc()` family of dynamic memory allocation functions. This case is significant because the allocator manages the heap, a resource shared among all threads of the application. As we saw in Section 2.1, naïve attempts to provide asynchronous cancellation often corrupt the heap if they interrupt the allocator. It is even unsafe to call into the allocator from a thread that has only been asynchronously paused while allocating, as it can cause a deadlock on the locks intended to protect the heap from concurrent access by different threads. This is the reason why signal handlers are not allowed to allocate memory. One way around these problems would be to use a separate heap for each preemptible function, but we have avoided this because it would complicate the ownership of objects that are allocated by a preemptible function but escape its scope before it terminates.

Instead, we consider the interfaces to the dynamic allocator to be **uninterruptible** functions. Although each libset contains a separate copy of them, all except one are inactive. Specifically, we route all calls to uninterruptible functions back to the **starting libset**, the set of modules loaded before *libgotcha* loaded any additional copies. The set of uninterruptible functions is currently governed by an internal allowlist within *libgotcha*. During load time, the constructor transcribes this information into the shadow GOTs so lookups will incur no additional runtime overhead.

Although the next libset does not determine which copy of an uninterruptible function gets invoked, it is relevant during the call. To avoid creating a dependency between the starting libset and the current one, it is important that the next libset be set to the starting libset while any uninterruptible code is running. To preserve this invariant, whenever `procedure_linkage_override()` detects that any libset other than the starting one is calling an uninterruptible function, it resets the next libset to the starting libset, storing a backup of the previous value. Then, just before invoking the function, it pushes the address of a trampoline function onto the stack. When the function eventually returns, this trampoline runs; before transferring control back to the call site, it restores the next libset.

[6] The faulting address would not be found if the program was dereferencing at an offset into the global, but one would just return the next-lowest entry.

In addition to controlling *libgotcha*'s treatment of calls to uninterruptible functions, the next libset also communicates valuable information to the control library. The starting libset must always be valid, and because we maintain the invariant that the next libset is always equal to the starting libset when the current libset is and third-party code is executing,[7] it is safe to preempt execution if and only if the next libset is not equal to the starting one. It is crucial that the control library check this before deciding to preempt, and as we saw in Listing 3.2, it can do so using the `libset_thread_get_next()` function.

Unlike function calls, accesses to global variables do not have defined end points after which they are complete. That is, a single write might leave a structure in a temporarily-invalid state that is corrected by a subsequent one, but there is no structural association between the two. We do not currently change the next libset when we resolve a global variable access, which means a control library might pause or cancel code in the middle of such a write or sequence of writes. This is safe because the allowlist does not contain any writeable variables, so the only time that a global variable write can resolve to a definition in a different libset is when the current libset is the starting one, the next libset is any other one, and the program performs an access such as (1) from Listing 3.1. Such an access would be routed to the code's own libset, so any corruption due to concurrency would remain properly isolated and could not infect the starting libset.[8]

### 3.6.1 Other uninterruptible functions

The memory allocator interfaces are not the only functions in the allowlist. The dynamic linker behaves specially with respect to namespaces: although it appears to be loaded into every namespace, it refuses to load additional copies of itself and instead includes special logic that proxies calls to it from other namespaces back to the main one. If such calls were considered interruptible, the proxying would change the current libset out from under *libgotcha*, violating the invariant that the next libset must equal the starting libset when the current one does and potentially corrupting the dynamic linker. To avoid this, we add all of the dynamic linker's functions to the allowlist.

The dynamic linker introduces other complications as well. While itself dependency free, the GNU implementation is part of the glibc project, and other glibc modules depend on it, including on internal interfaces that should not be exposed to the rest of the program. To keep these interfaces private, the dynamic linker does not export them as dynamic symbols, and instead exposes them via an opaque data region whose layout is partially known to the other glibc modules. Fortunately, the region is split into separate subregions under the `_rtld_global_ro` and `_rtld_global` dynamic symbols based on whether the area is writeable, which helps us determine whether a particular use has the potential to corrupt the dynamic linker state. In particular, we have allowlisted all the functions that access the latter structure, a set currently consisting of `fork()`, `posix_spawn()`, `uselocale()`, and `__cxa_thread_atexit_impl()` from `libc.so`

[7] By third-party code, we mean code that does not interact with the selective relinking primitive or any abstraction built on top of it. Any code that makes use of selective relinking is responsible for doing so safely (i.e., ensuring that it does not corrupt its own state).

[8] If one did need support for allowlisted writeable global variables, the resolver signal handler described in Section 3.5.2 would need to serve a read-only copy of such variables; this way, each write would generate an additional segmentation fault and the handler could batch writes to the backing object. Determining a safe boundary for this batching would require at least additional runtime heuristics, if not static analysis.

and `pthread_create()` from `libpthread.so`. To flag potential breakage from future changes in the region's use, *libgotcha* emits a warning at load time whenever it encounters an unexpected access to `_rtld_global`.

We have encountered one instance where a glibc module modifies `_rtld_global` without a corresponding dynamic function call. The libpthread constructor monkey patches the dynamic linker to replace pointers to stubbed-out mutex functions with its own proper implementation before the application can spawn any POSIX threads.[9] These changes would create a hazardous dependency between the dynamic linker and whichever copy of libpthread was last loaded (in our case, the one in the last libset). To avoid this problem, *libgotcha*'s constructor checks for libraries that (1) directly access `_rtld_global`, (2) have the type of constructor in question, and (3) are marked with the ELF configuration flag `DF_1_NODELETE` to indicate that they are unsafe to unload once present in the process image. Before loading copies of a library with all of these properties into any libsets, we create a temporary copy of its shared library with its ELF metadata tweaked to prevent the constructor from running. We add these patched libraries to the beginning of the search path so libsets do not modify the dynamic linker at load time and it continues to depend on the copy of libpthread in the starting libset.

In a few places, a glibc module asks the dynamic linker to load or unload another shared library at runtime by calling the internal `_dl_open()` or `_dl_close()` interface using a function pointer hidden in `_rtld_global_ro`. The most obvious of these is the public-facing functions in `libdl.so`, the same interface that *libgotcha* uses to populate libsets. We considered adding all of this library's functions to the allowlist, but this presents a problem: in order to decide which namespace to load the library into, `dlopen()` checks which module contains its return address.[10] Treating it as uninterruptible would result in a libset switch when calling it from outside the starting libset, pushing a *libgotcha* trampoline "return" address onto the stack and thereby causing all dynamically-loaded libraries to be added to the starting libset instead of the current libset. Other functions that use these internal dynamic loading calls include the `iconv()` family of character-conversion functions, `getaddrinfo()` and the other DNS-translation functions (which can load GNU Libidn to handle internationalized domain names), and the modular Name Service Switch system for accessing the users, hosts, protocols, and services databases. Adding these functions to the allowlist would not break them, but it would treat what are potentially long-running operations (even including network communication) as uninterruptible code. We instead handle both of these situations by replacing the `_dl_open()` and `_dl_close()` pointers in `_rtld_global_ro` with our own hook functions that transition to uninterruptible code only while modules are being loaded or unloaded. Since the region is opaque and its layout is subject to change and dependent on the glibc build configuration, we find the appropriate pointers by repeatedly replacing one entry at a time with a probe function and executing no-op `dlopen()` and

[9] It also installs hooks to support stack execution protection and enable full support for thread-local variables. As with the mutex changes, these tweaks appear to replace stubbed placeholders rather than establish information flow from the dynamic linker to the specific copy of libpthread.

[10] Note that this means a `dlopen()` loads the module into the current libset rather than the next one. We preserve this standard system behavior even though it is somewhat surprising. The only time it could cause unexpected results is when a module containing a preemptible function definition loads other modules at runtime, and since it used preemptible functions, such a module could be expected to be aware of this case. If this choice later proved problematic to application designers (e.g., of systems that make heavy use of runtime plugins), it would be possible to instead base the destination on the next libset by injecting a trampoline return address from the appropriate copy of *libgotcha*.

```
// Pointer to function taking and returning void
typedef void (*libset_cb_t)(void);

void libset_register_callback(libset_cb_t);
void libset_register_returnback(libset_cb_t);
libset_t libset_of_caller(void);
```

**Listing 3.3:** *libgotcha* C callback interface

`dlclose()` operations until the probe function gets called. Once we have found both pointers, we record their original values, mark their containing page writeable, replace them with pointers to our hook functions, and restore the page protections.

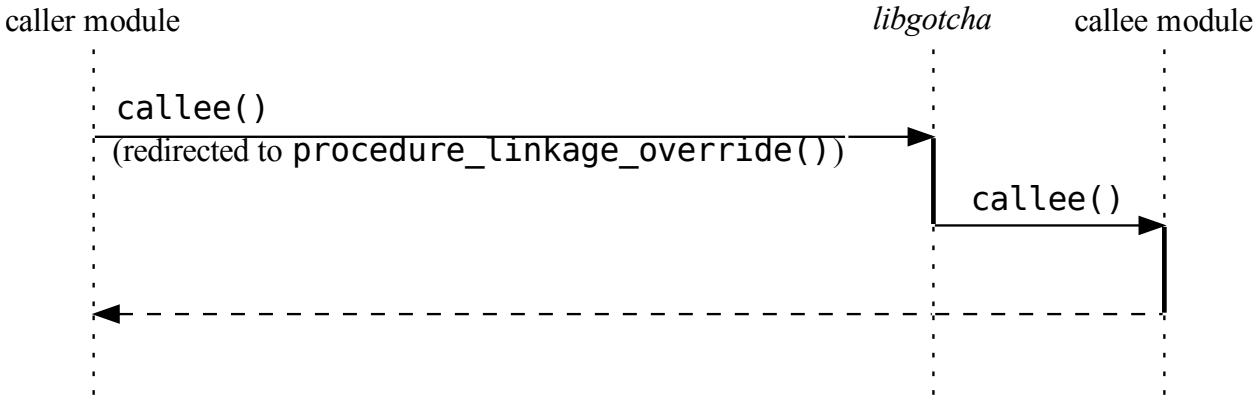
### 3.6.2 Control library callbacks

While a control library can identify an uninterruptible task by observing that the thread's next libset is equal to the starting one, many control libraries will benefit from active notification of interruptibility. For this purpose, *libgotcha* provides a callback interface that allows the control library to register functions that should be invoked whenever an uninterruptible function is called or returns. A control library can use such **call callbacks** and **return callbacks** to disable and reenable preemption mechanisms or establish a critical section around uninterruptible code (e.g., by taking a mutex or blocking signals). Callbacks run when we automatically switch to the starting libset from any other; since this change is idempotent, a callback is only invoked once at the beginning or end of each uninterruptible region, even if the function calls others within the starting libset. Figure 3.5 shows the effect of such callbacks on function call interception by comparing against both interruptible calls and uninterruptible calls when no callbacks are registered.

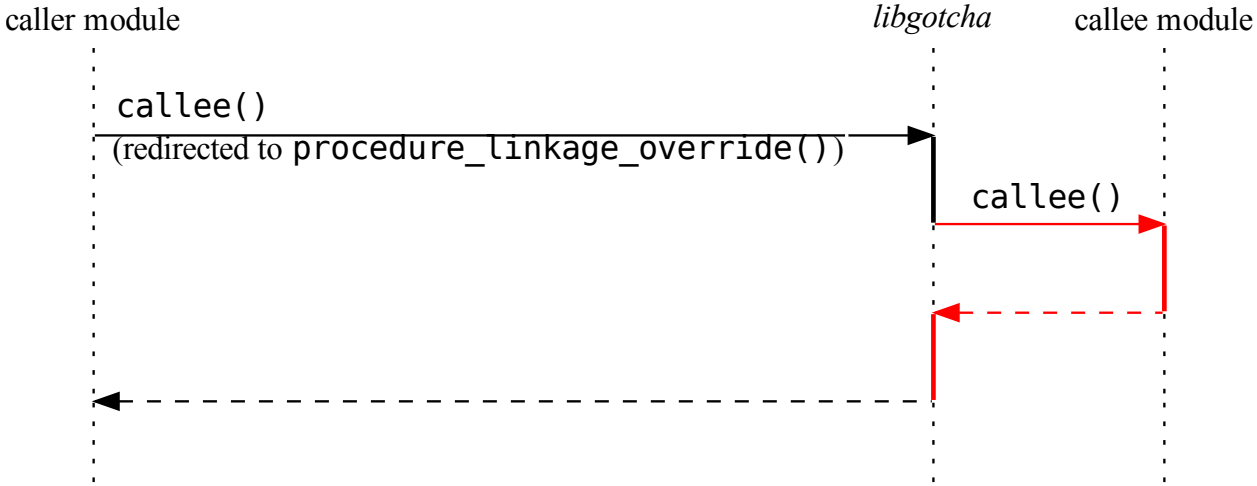
Listing 3.3 shows the functions for managing callbacks, which form an extension to the main *libgotcha* control API. The control library can pass a function pointer to either of the registration functions to have it invoked on entry to or exit from uninterruptible code. In developing this interface, we discovered it was very hard to write correct callbacks that could tolerate being preempted, so we elected to run all callbacks with the next libset set to the starting one.[11] Recall that it is never safe to pause or cancel execution in this state, so the control library's preemption mechanism will defer preemption in callbacks, just as it was already required to do in other such places. Because the control library's callback might want to know which libset control switched from, we provide a `libset_of_caller()` function for retrieving the previous value of the next libset (which is also the value it will be restored to when exiting the uninterruptible region).

The way that we currently implement callbacks places some restrictions on what they

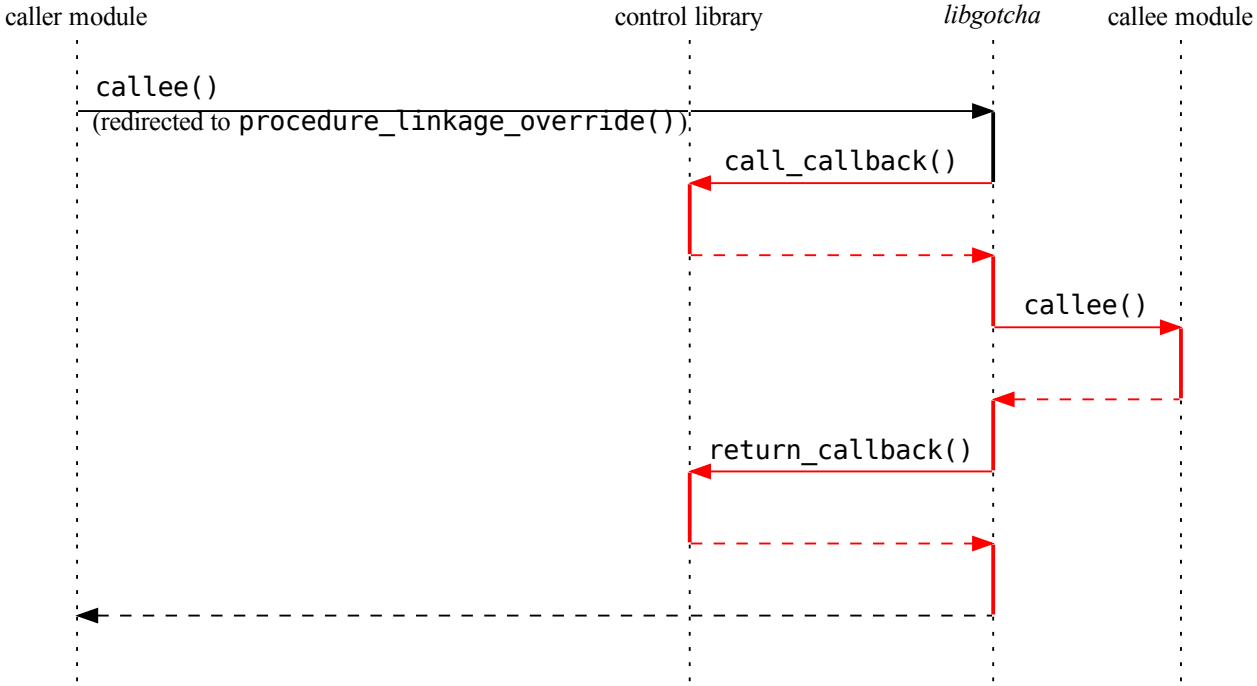
[11] This is actually only currently true of the call callback, as return callbacks run in the caller's libset for largely historical reasons. We originally made this decision because this callback type predated the mechanism for querying the caller's libset. After adding both this feature and call callbacks, we planned to change the behavior for consistency between the two callback types, but encountered a complication. Some publicly-callable and initially uninterruptible *libgotcha* functions sometimes need to change the next libset partway through in order to finish in an interruptible state, and we did not want such changes to be observable by the callback.




**(a)** Interruptible call




**(b)** Uninterruptible call when no callbacks are registered




**(c)** Uninterruptible call when callbacks are registered for both calls and returns

**Figure 3.5:** Interception of cross-module function calls. Solid lines represent function calls; dashed ones represent returns. Color marks uninterruptible code (i.e., next libset = starting libset).

are allowed to do. The *libgotcha* assembly trampoline that invokes return callbacks backs up the integer return registers, but such callbacks cannot use floating point unless they manually save and restore the floating-point return registers. As an implementation shortcut, `procedure_linkage_override()` invokes the call callback via a `SIGTRAP` handler by executing an `int3` instruction, in order to avoid having to save and restore a large set of registers itself (since the return registers account for only two of the many caller-saved registers). The signal occurs at a well-known point in execution (e.g., not in the middle of memory allocation), so it does not subject the callback function to the usual safety constraints on signal handlers, but it does make this type of callback orders of magnitude slower. Fortunately, we anticipate rarer need for them; *libinger*, for example, only uses a return callback.

## 3.7 Control libraries

Selective relinking is a low-level abstraction useful in special situations that require fine-grained memory isolation (e.g., legalizing types of concurrency that would otherwise exhibit undefined behavior). As such, *libgotcha* takes no stance on how libsets should be applied, instead leaving it up to other code to use them (by changing the next libset). We refer to this other code as a control library because it controls *libgotcha* using the API shown in Listings 3.2 and 3.3.

The semantics of a control library depend on how it is built:

- An **internal** control library is statically linked with *libgotcha* to form a single shared object file that is loaded (or preloaded) as one unit. Because each libset contains a separate copy of its module, *libgotcha* adds itself to the uninterruptible allowlist described in Section 3.6. This redirects all calls to it into the starting libset, rendering its other copies dormant in a manner similar to the dynamic linker's proxying mechanism. By virtue of being part of *libgotcha*'s module, internal control libraries enjoy the same guarantee that their code always executes in the base libset.[12] This also means that they, like *libgotcha*, are automatically considered uninterruptible.

- An **external** control library is dynamically linked with *libgotcha*, and therefore constitutes a separate shared object file that depends on `libgotcha.so` (or, perhaps, a compatible internal control library). Calls into such a library do not cause an automatic libset switch unless explicitly allowlisted within *libgotcha*, so the library is responsible for coping with preemption and multiple copies of itself. Despite these limitations, support for external control libraries enables control libraries to compose without mutual build system support, allows a single application to have more than one unrelated control library, and supports updating to a new version of *libgotcha* without rebuilding the control library.

[12]There is actually one exception to this rule: control libraries written in Rust can include generic functions in their public interface. Because rustc monomorphizes such functions for the client code that uses them, *their implementations are not actually in libgotcha's module!* Rather, there are one or more copies of them (specialized for various type arguments) in each program module that calls them. It is difficult to write correct control library code that can tolerate preempting itself or interacting with other copies of itself, so we recommend that control libraries with generic interfaces manually transfer control to the starting libset before doing the bulk of their work. One way to improve the experience for control library authors would be to provide a procedural macro compiler plugin that would at least partially automate this using custom Rust attributes.

### 3.7.1 Enforced interposition

A few libc functions either present an opportunity to circumvent the libset abstraction or misbehave when called from a program that uses libsets. To address this, *libgotcha* uses a dynamic-linking trick called **interposition** to substitute its own implementation of these functions. Interposition occurs when one module defines a symbol with the same name as another module's symbol, resulting in calls intended for the other module being routed to it instead. This relies on the interposing library appearing earlier in the dynamic linker's dependency search order, so the technique is only reliable when the application binary is the interposer or the interposing library is preloaded by defining the `LD_PRELOAD` environment variable before executing the application.

While *libgotcha* supports being preloaded, it is intended to work seamlessly even when the application is launched normally, so normal interposition is not strong enough. Fortunately, we already control dynamic linking by rewriting GOT entries, so at the same time, we implement a variant that we term **enforced interposition**. Whenever we encounter a relocation in a third-party module referencing a dynamic symbol with the same name as one defined in *libgotcha*'s module, we replace its GOT entry with the address of the PLOT stub corresponding to that definition, so that all calls to that symbol from any other module are redirected to *libgotcha*.[13] Conversely, whenever we encounter such a dynamic symbol that is also referenced from *libgotcha*'s module, we replace its GOT entry there with a reference to the real third-party definition (or interposition, as the case may be). In this way, the code in *libgotcha* can implement wrappers around standard library (or any other) functions without inadvertently interposing its own uses thereof.

We saw in Section 3.5.2 that one fraught case is the `sigaction()` function, which could be used to replace the *libgotcha* signal handler that resolves global variable accesses at runtime to the appropriate definition under the rules of selective relinking. As such, we provide a replacement implementation that checks whether the caller is trying to install a `SIGSEGV` handler; if so, instead of honoring the request, we store the handler internally and our handler calls it whenever it is unable to recover from a segmentation fault by fixing one of our fake addresses. We provide a similar replacement for the legacy `signal()` function. One other signal-related problem can occur when an application or library temporarily blocks signals to establish a critical section. Blocking our handler would cause the critical section to crash if it accessed a global variable, so we replace `sigprocmask()` and `pthread_sigmask()` to leave `SIGSEGV` alone. Finally, we replace `sigfillset()` and `sigaddset()` to exclude `SIGSEGV`, which prevents signal handlers from implicitly masking and unmasking it (via Linux's `sigreturn()` helper function[14]). The latter replacements are simple enough to make good examples, so we include them in Listing 3.4. Note that although they may appear recursive, they are not: enforced interposition rebinds their references to their own names to the external definitions in `libc.so`.

Another troublesome call is `dlsym()`, used to manually look up the address of a dynamic symbol by name. For the same reason we encountered with `dlopen()` in Section 3.6.1, simply marking it as uninterruptible would cause it to search the wrong libset. Because it implements its logic directly in `libdl.so` rather than deferring to a private dynamic linker interface, we

[13] We export a minimal and deliberate subset of *libgotcha*'s functions as dynamic symbols to avoid unexpected applications of this rule.

[14] This function is provided by the kernel for direct use by unprivileged code. It is defined in the VDSO (Virtual Dynamic Shared Object), an emulated shared library that the kernel maps into each processes's address space.

```
int sigfillset(sigset_t *set) {
        int res = sigfillset(set);
        if(!config_noglobals())
                sigdelset(set, SIGSEGV);
        return res;
}

int sigaddset(sigset_t *set, int signum) {
        if(!config_noglobals() && signum == SIGSEGV)
                return 0;
        return sigaddset(set, signum);
}
```

**Listing 3.4:** *libgotcha* `sigfillset()` and `sigaddset()` replacements

initially assumed it would work fine out of the box; however, when we implemented the warning about direct accesses to `_rtld_global` described in that section, we were surprised to find that `dlsym()` triggered it. Some investigating revealed that `dlsym()` was directly taking the dynamic linker's big lock for the entire duration of every lookup because, whenever it hands one module the address of a symbol defined in another, it adds a dependency edge between them to ensure the defining module cannot be unloaded before the referencing one.[15] To solve this and prevent many manual symbol lookups from being uninterruptible, we implemented a `dlsym()` replacement that becomes interruptible (by restoring the next libset) and does not take the big lock when it detects that the calling code's current libset is not the starting one and that it is looking up a symbol in some specific other module in that libset. This is safe because such lookups can only modify the dynamic linker state corresponding to their own libset, and the state changes (or any breakage caused by pausing or cancellation) will therefore not impact any other libsets.[16] Our replacement function also makes one important tweak to the symbol lookup semantics: if we have a PLOT entry or fake global address for the requested symbol, we return that instead of its true definition to avoid creating a backdoor around selective relinking and to preserve pointer comparison. As part of these changes, we also had to add a `dlerror()` replacement to report errors encountered during symbol lookups.

[15] Technically, there is one other reason for taking the big lock: to prevent a concurrent `dlclose()` from removing the defining module and/or corrupting the dependency graph while the symbol search is in progress. Unloading modules is a rare operation that we currently consider out of scope, so we do not guard against this situation. However, one feasible approach would be to replace `dlclose()` with a version that took a *libgotcha*-local lock to establish mutual exclusion with symbol lookups without taking the big lock. To avoid making symbol lookups uninterruptible, `libset_reinit()` could break the local lock in response to cancellation. A simpler option would be a `dlclose()` replacement that disallowed unloading modules from libsets other than the starting one; this would have the additional benefit of preventing a preemptible function from making a lasting change to the modules available in its (reusable) libset.

[16] We currently remain interruptible if it is traversing the entire global search list rather than searching a specific module, because the symbol might resolve to a definition in the dynamic linker. In this case, the dynamic linker would add a dependency edge into the starting libset because of the dynamic linker's special proxying. We suspect that such searches could safely be made preemptible if one checked the symbol being searched against the dynamic linker's symbol table to ensure this outcome was impossible.

Finally, we encountered one apparent incompatibility between exception handling and dynamic linker namespaces. After noticing that Rust panics would crash the program when the current libset was not the starting one, we discovered that libgcc and libunwind's implementations of the `_Unwind_RaiseException()` function from the C++ ABI (Application Binary Interface) call glibc's `dl_iterate_phdr()` function to find each module's `.eh_frame` section containing metadata about the stack frame address. We then noticed that the latter function only searches the invoking namespace, which causes unwinding to fail as soon as it encounters a function call that crossed a namespace boundary. We fixed this by adding a replacement implementation that extends the search to other namespaces as long it has not yet found the module the caller is seeking.

Because internal control libraries are within the same module as *libgotcha*, enforced interposition treats them in the same special way. This enables such control libraries to define their own replacements merely by implementing a function with a well-known name and exporting it in their dynamic symbol table. By default, all of their own uses of such a symbol refer to the third-party definition. We recognize that control libraries may need to further wrap our replacement functions instead of overriding them completely, however, so for each symbol *name* that *libgotcha* implements a replacement for, it also exports a symbol `libgotcha_`*name* that refers unambiguously to that replacement. These symbols are local by default to prevent abuse, but any control library that needs them can configure its build system to change their type before linking against the *libgotcha* static library.

### 3.7.2 Thread-local storage

Selective relinking seeks to isolate global state that would otherwise infect the entire program if left corrupted or inconsistent by pausing or cancellation. We saw in Section 3.5 that this state includes global variables and functions (which might be nonreentrant). However, there is a third class of state that escapes a function's scope: thread-local variables.

Thread-local variables reside in a memory region known as the TLS (Thread-Local Storage). This, in turn, is located just before the TCB (Thread-Control Block), a region storing metadata about the thread (Figure 3.6). The pthreads implementation maintains a pointer to the current thread's TCB in the `%fs` segment register for fast access, so the compiler simply translates references to thread-local variables into negative offsets into that segment.[17]

Unlike the types of state we have seen so far, the desired semantics of thread-local variable accesses depend on how the control library is using libsets. For some use cases, it may even be necessary to widen the scope of thread-locals *beyond their thread*; for instance, this is required to support pausing code on one kernel thread and then resuming its execution on a different one. Furthermore, there is no efficient way to consistently make relinking selective based on the module relationship between the referee and the referent because TLS pointer calculations do not use GOT entries and can use hardcoded offsets. But neither is it clear whether the abstraction provided by any given control library can tolerate a thread-local variable's apparent value changing

[17] This is a slight simplification because the layout of the TLS depends on the order in which libraries are loaded, and therefore is not fully known until *runtime*. Depending on the thread-local variable's scope and which module it is defined in, the dynamic linker may have to determine the offset in the middle of program execution. Each combination of these two factors constitutes a different access "model," and causes the compiler to emit a different sequence of instructions to find the variable.

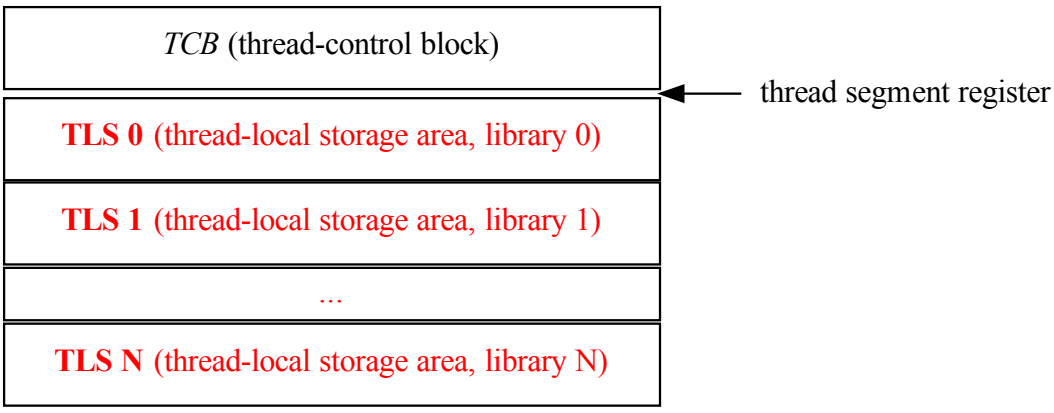



**Figure 3.6:** Per-thread portion of process image

as soon as the next libset does and while the current libset remains the same, which would make `thread_local` variables fail assertions analogous to (2) and (3) from Listing 3.1.

Rather than being opinionated and choosing the wrong stance, *libgotcha* leaves the scope of thread-local variables up to the control library. The default is the system behavior, where thread-local variables are associated with the kernel thread and unaffected by the next libset (though of course they do depend on the current libset, since the TLS contains a separate copy of each thread local for each copy of its defining module). However, if the control library instead wants to tie the scope of "thread"-local variables to that of a custom task abstraction, it may allocate custom TCBs and swap their pointers into the segment register when switching tasks. One hazard for this approach is that `pthread_self()` simply retrieves the TCB pointer from the thread segment register, so changing it alters the thread's view of its own identifier. The most significant consequence of this is that if the thread attempts to send itself a signal (e.g., to support preemption), the thread ID it passes will be invalid and the operation will fail. To address this, we provide replacement functions for the `arch_prctl()` function for updating segment registers[18] and the `pthread_kill()` function for sending signals directed at a specific thread. Respectively, our replacements store the pointer to the original TCB the installed one is replacing and substitute the original pointer when they see one saved.

One thing that *libgotcha* does have to do is restore the TLS upon a cancellation, since failing to do so might leave it permanently corrupt. A naïve approach would be to do this for every thread's TLS, but this could be quite expensive, especially since it would require thread synchronization. Instead, we observe that each libset is likely to only ever be used by a single thread, and that threads that never use a particular libset will never reference their TLS areas that correspond

[18] Although x86-64 processors provide an instruction for updating the segment registers, the operating system can disable its use by non-supervisor code. The Linux kernel does so to guard against a class of privilege-escalation bugs, and instead provides a system call for changing segment registers so that it can validate the new location before applying the update. While a rogue application could avoid our replacement function by calling `syscall()` instead of the glibc wrapper function, this is a rare case and a difficult one to guard against.

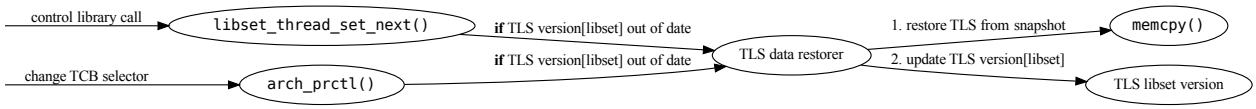



**Figure 3.7:** Lazy TLS reinitialization following asynchronous cancellation and libset reuse

to it. Consequently, we reinitialize TLSes lazily: We allocate a thread-local variable (per TCB, of course) to track each TLS's current "version" of each libset. Whenever a control library calls `libset_reinit()` to signal a cancellation, we merely increment a separate program-wide version watermark for that libset, as shown at the bottom of Figure 3.3. We know that a libset whose execution has just been cancelled cannot currently be in use by any thread, so we do not need to do anything further unless and until the libset is reused. We can recognize such reuse because it begins with the control library passing the libset identifier to `libset_thread_set_next()`, and we respond by checking the current TCB's current version for that library against the watermark. If the TCB is out of date, we iterate over the TLS's regions and restore the contents corresponding to modules from that libset using each module's initialization image in its `.tdata` and `.tbss` sections. Then we update the TCB's current version for that libset to match the watermark. Of course, the control library might have multiple TCBs that it uses with the same libset, but we detect this by following the same procedure from our `arch_prctl()` replacement function. Figure 3.7 summarizes the algorithm; note that it is even robust to control libraries that use the same libset from multiple threads concurrently, as long as they never illegally reinitialize a libset while it is in use by any thread.

It is important that if a control library switches the TCB, it does not affect *libgotcha*'s internal state. Yet *libgotcha* relies on thread-local variables to store information such as the pointer to the thread's original TCB. In addition to the enforced dynamic interposition mechanism described in the previous section, *libgotcha* includes a static interposition system whereby it can define statically-linked replacement functions (not exported to the dynamic symbol table) that apply throughout its own code but not to the rest of the program, even internal control libraries. This works because the static linker prefers to link against static symbols provided directly in object files so that it can avoid generating relocations against the GOT, but given the choice between (static) symbols in a static library and (dynamic) ones in a shared library, it prefers to link against the latter (so for internal control libraries, the third-party definition rather than *libgotcha*'s static replacement). To insulate its own thread-local variables against TCB changes, *libgotcha* includes a static replacement for the dynamic linker's `__tls_get_addr()` function that reroutes all its accesses to its own thread locals to the original TCB for the current thread.[19]

[19]There are currently three thread locals that *libgotcha* deliberately accesses via static offsets from the TCB instead of using `__tls_get_addr()`. The first stores the current version counters for each libset, and is truly specific to the TCB rather than the kernel thread. The other two still correspond to the kernel thread, but are accessed from assembly by `procedure_linkage_override()` and its return trampoline, whose implementation is greatly simplified by being able to access them without calling into C code. One is the thread's record of the next libset before the libset switch, which our `arch_prctl()` replacement manually copies over whenever the control library swaps in a different TCB. The other is the actual next libset, which is always set to the starting libset while a TCB swap occurs because of said function replacement, and the trampoline will automatically restore its value from the aforementioned upon return from the uninterruptible call.

## 3.8 Limitations

Selective relinking successfully mitigates the shared state problem to allow safe forms of asynchronous pausing and cancellation on top of the existing systems stack. That said, we should emphasize a few shortcomings of our approach:

### 3.8.1 Portability

Selective relinking is grounded in general principles of dynamic linking and the ELF specification. The main burden of porting *libgotcha* to another architecture would be rewriting the assembly portions (`procedure_linkage_override()`, its trampolines, and the PLOT stubs), which currently account for under 300 lines of code. However, porting to another operating system would be more difficult, as selective relinking requires linker namespaces, which are only available natively on Solaris derivatives and the GNU system (although modern versions of Android's Bionic runtime appear to include internal dynamic linker features sufficient to support the namespace abstraction [4]). And of course, selective relinking is only relevant to dynamically-linked applications, which precludes its use in embedded systems or with those programming languages that do not bother to support dynamic linking.

Furthermore, getting *libgotcha* to run real unmodified applications has required handling many GNU/Linux-specific features and behaviors. Examples include indirect function calls (Section 3.5.1); dynamic linker proxying, glibc-wide shared state, monkey patching, and implicit module loading (Section 3.6.1); `sigreturn()` signal masking, big locks that escape the dynamic linker, and the interaction between namespaces and exception handling (Section 3.7.1); and thread-local storage details such as TCB switching and `__tls_get_addr()`'s interface (Section 3.7.2). As such, porting to a different dynamic linker and C library would surely require a fair amount of systems hacking and debugging.

Indeed, *libgotcha*'s degree of dependence on glibc means that even upgrades can require some work. We originally developed *libgotcha* atop glibc 2.29, and we found that both the 2.30 and 2.31 releases introduced breaking changes.[20] Fortunately, we have prioritized debuggability throughout the development of *libgotcha*, and provide several features to make it less painful. Among these are a shell script for tracing intercepted function calls and global variable accesses, a GDB script that automatically loads debugging symbols for namespaces other than the starting one, environment variables for disabling features that make debugging difficult (e.g., global variable interception, which generates numerous segmentation faults), and specific workarounds to support running under Valgrind [60] and Mozilla's rr reverse debugger [49]. With the help of these tools, we were able to port to both glibc versions; furthermore, *libgotcha* subsequently survived

[20] The 2.30 release stopped accepting position-independent executables as arguments to the `dlopen()` family of functions because they can contain `COPY` relocations that break if they are not present from the start of the program. As we discuss later in this section, we do not support such relocations anyway. However, we need to open a copy of the executable in each libset just like any other module, so we work around the change by stripping the `DF_1_PIE` ELF flag from the executable using the same approach we use for monkey-patching constructors (Section 3.6.1). The 2.31 release introduced a dependency between the number of supported namespaces and the size of the `_rtld_global_ro` structure. At hundreds of libsets, this caused the member to grow as large as 16 pages, triggering the symbol size limitation discussed at the end of Section 3.5.2. We responded by increasing the (currently hardcoded) number of inaccessible pages used to intercept global variable accesses.

the 2.32 and 2.33 upgrades unmodified.[21]

### 3.8.2 Scalability

Although *libgotcha* is compatible with an unmodified glibc in principle, vanilla glibc builds are limited to 16 dynamic linker namespaces including the main one. This means that we can only create up to 15 new libsets, and that a control library can only have this many preemptible tasks running and/or paused at any time. To raise this limit, we rebuild the glibc sources with an increased value of the `DL_NNS` macro, which controls the number of supported namespaces. There is no need to install the resulting runtime on the host system; instead, we supply a custom interpreter path when linking any application that needs more libsets. We have tested this configuration up to 512 namespaces, but the setting remains a fixed (glibc) compile-time limit. Removing the per-process libset limit altogether would require porting *libgotcha* to work with an alternate dynamic linker that allocates namespaces at runtime, such as drow [17].

Relatedly, *libgotcha* itself prepares all libsets at load time, and therefore delays program startup in proportion to both the number of libsets and the number of modules. The good news is that doing this work up front significantly reduces the latency at runtime. That said, initializing libsets at runtime would allow control libraries to strike their own balance between startup time and runtime cost. Selective relinking is not fundamentally incompatible with such a feature, but supporting it would require significant engineering effort, most notably modification of all existing data structures to support resizing.

Initializing more than a few libsets reveals another scaling limitation in glibc: certain dynamic libraries include eager relocations that require their thread-local variables to be assigned TLS offsets at load time. To support this, the GNU dynamic linker reserves a certain amount of static TLS space that must be sufficient for all libraries with this requirement across all namespaces. Populating the libsets multiplies modules' space requirements, and can quickly exhaust this static area. Part of the problem is that *libgotcha* itself includes such relocations to make its assembly portions easier to write, and since it currently loads copies of itself into each libset, it too contributes to the increased footprint. When launching a program that must support more than a few libsets, one must export an environment variable to tune glibc's static TLS size.

### 3.8.3 Flexibility

Selective relinking supports all but one of the cross-module dynamic relocation types. All forms of `GLOB_DAT` relocations work: global variables, eagerly-resolved function calls, and indirect function calls. Lazily-resolved function calls via `JUMP_SLOT` relocations work as usual (including memoization), except that we must eagerly resolve some of them to determine whether they represent cross-module calls.

All TLS access modes are supported by swapping out the TCB, although the semantics of thread-local accesses depend on the control library's choice of when to do so. Unlike with global

[21] More recently, glibc 2.34 has consolidated the interfaces formerly exposed by `libpthread.so` and other files into a monolithic `libc.so`. As of this writing, we have not tried to run *libgotcha* atop this version, but we expect this major restructuring to require at least minor changes. We suspect the problem is not unique to us: In the more than six months since the upstream release, most major GNU/Linux distributions have not yet stabilized this version. This includes distributions such as Arch and Gentoo that are typically known for shipping bleeding-edge packages.

relocations, selective relinking does not reroute cross-module accesses based on the next libset, so the control library has the choice between resolving all thread-local variable accesses to their definition in either the current libset or the next libset at any given time. While either of these choices might be confusing, thread-local "globals" are quite rare. Of the three such variables remaining in glibc's `libc.so`, only `errno` was ever part of its stable interface; even it has been obsoleted, with the Linux Standard Base long specifying that it is to be defined as a preprocessor macro invoking the `__errno_location()` function [44]. The other two are used for communication between glibc modules, and hold ephemeral state that needs only persist while code is conducting associated calls to the C library's DNS-resolution facilities, something that is unlikely to be split up regardless of the control library's choice of "thread" isolation boundary.

The only cross-module relocation type we do not support is the `COPY` relocation, which exists as an optimization to accelerate the executable's access to global variables defined in other modules. It works by having the static linker allocate redundant space for all libraries' globals directly within the executable's file, then asking the dynamic linker to initialize them at load time by copying the contents of the version in their defining modules. The dynamic linker then sets all other modules' GOT entries to refer to the new copies, thereby rendering the libraries' own definitions vestigial. Because the definitions are all located within the executable's module, it can access them directly using instruction-relative offsets rather than looking up their addresses in the GOT. However, this transformation breaks selective relinking for three reasons: (1) The elision of GOT entries from the executable means that we can no longer intercept the executable's cross-module accesses to global variables to redirect them to the next libset, causing Listing 3.1 to fail assertion (1) if its code is located in the executable. (2) In performing the setup work, the dynamic linker assumes that only one copy of the executable will ever be loaded, an assumption that the libset abstraction violates. (3) Migrating the effective definitions from libraries' modules into the executable's disassociates symbols from their defining library, causing Listing 3.1 to fail assertion (3) if its code is located in a library. When building a program that depends on *libgotcha*, programmers must instruct their compiler to disable `COPY` relocations, as with the `-fpic` switch to GCC and Clang. If *libgotcha* encounters any `COPY` relocations at load time, it prints a warning that the application is unsupported as compiled. Forsaking `COPY` relocations does incur a small performance penalty, but exported global variables are rare now that thread safety is a pervasive concern in system design.

## 3.9 Evaluation

We benchmarked *libgotcha* on an Intel Xeon Gold 6130 (Skylake) server clocked at 2.1 GHz and running Linux 5.4, rustc 1.56.0, gcc 9.3.0, and glibc 2.33.

### 3.9.1 Microbenchmarks

Recall that linking an application against *libgotcha* imposes additional overhead on most dynamic symbol accesses; we report these overheads in Table 3.1a. Eager function calls account for almost all of a program's dynamic symbol accesses: lazy resolution only occurs the first time a module calls a particular function (Section 3.1.2) and globals are becoming rare (Section 3.8.3).

Table 3.1b shows that the *libgotcha* eager function call overhead of 15 ns roughly doubles the

| Symbol resolution scheme | Time without *libgotcha* (*ns*) | Time with *libgotcha* (ns) |
|---|---|---|
| eager (load time) | 1 ± 0 | 15 ± 0 |
| lazy (runtime) | 95 ± 2 | 110 ± 3 |
| global variable | 0 ± 0 | 4917 ± 46 |

**(a)** Generic symbols, without and with *libgotcha*

| Baseline | Time without *libgotcha* (ns) |
|---|---|
| `gettimeofday()` | 17 ± 0 |
| `getpid()` | 361 ± 16 |

**(b)** Library functions and syscalls without *libgotcha*

| Trigger | Time with *libgotcha* (ns) |
|---|---|
| Uninterruptible call | 25 ± 0 |
| Uninterruptible call + call callback | 1813 ± 94 |
| Uninterruptible call + return callback | 26 ± 1 |

**(c)** Uninterruptible calls triggering a libset switch

**Table 3.1:** Runtime overheads of accessing dynamic symbols

latency of a trivial C library function (`gettimeofday()`) and imposes a less than 5% latency overhead on a simple system call (`getpid()`).[22] This overhead affects the entire program, regardless of the current libset at the time of the call. Additionally, calls to uninterruptible functions from outside the starting libset incur just under twice this latency to switch back to the main libset (Section 3.6); Table 3.1c shows this figure alongside the cost of notification callbacks (Section 3.6.2). Notification at the start of a call is markedly more expensive because of the implementation shortcut described in Section 3.6.2, but is unnecessary for preemptible functions.

### 3.9.2 Libset initialization and reinitialization

Because *libgotcha* front loads the work of populating libsets (Section 3.4) and updating GOTs (Section 3.5), it also inflates application startup time. To measure the extent of this effect, we preloaded *libgotcha* into version 1.17.0 of the Deno JavaScript runtime [14], which ships as a single 81-MB executable that includes the V8 engine [69] and all its other dependencies except for glibc and libgcc. We measured the execution time of an empty JavaScript program, a good proxy for startup time because the script starts running mere milliseconds before the process terminates. We found that with only one libset enabled, selective relinking raises the execution time from 35 to 122 ms. Figure 3.8a shows how the runtime, memory footprint, and page faults scale with the number of available libsets.

The default Deno configuration provides a realistic benchmark of *libgotcha*'s impact on a large application, but we also wanted to approximate a worst-case scenario. To do so, we rebuilt Deno

[22]The cost of a system call slipped by an order of magnitude in our benchmarks as a result of the Meltdown and Spectre mitigations, which require the kernel to unmap its page tables before context switching to user code to guard against timing attacks. Previously, a system call was approximately three times as expensive as the *libgotcha* eager function call overhead.

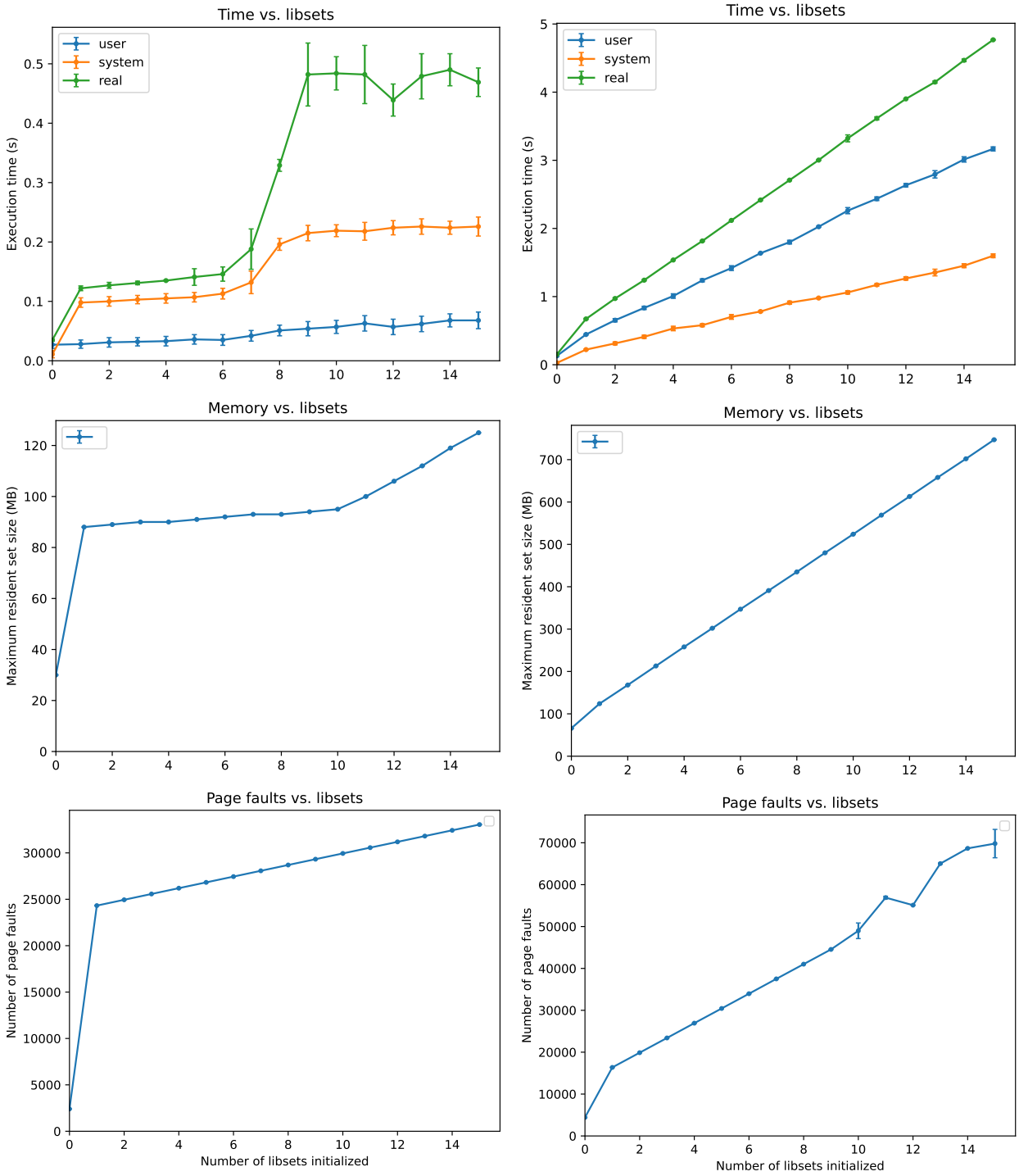
with V8 and each of its Rust dependencies compiled as a separate shared library, a configuration we will refer to as "Dyno." While Dyno's executable is only 14 MB, compared to Deno it depends on 240 additional shared object files whose size totals 268 MB. Dyno takes 4.4x as long to start without *libgotcha*, and selective relinking with a single libset slows it down 4.4x over that. We present its resource scaling behavior in Figure 3.8b.

We also measured the time to reinitialize a libset, as required after the cancellation of an isolated task. In Deno, this takes 167 µs, whereas in Dyno it takes 8039. Note that neither of the two configurations is ideal for actually making use of libsets: Deno statically links third-party libraries in the executable, meaning that their internal state will not be isolated. Dyno splits the build into much more granular linkage units than necessary, incurring noticeable latencies even without selective relinking. The most useful arrangement for selective relinking would be a balanced configuration, building only the executable's top-level dependencies—and any dependencies shared between those—as shared libraries. Such a system would experience *libgotcha* overheads somewhere between those for Deno and Dyno, but we expect they would be much closer to the former.

### 3.9.3 Thread spawn performance

While evaluating the latency of operations under *libgotcha*, we encountered a surprising degradation in thread spawn performance. Further investigation revealed that the duration of each `pthread_create()` was proportional to the number of libsets we had initialized. The culprit turned out to be our large static TLS footprint (Section 3.8.2). When allocating a TLS, glibc iterates over each module that uses static TLS space, `memcpy()`s its initialization image from the `.tdata` section of its module, and `memset()`s an area the size of its `.tbss` section. When spawning a thread, the new TCB and TLS are merged into the `mmap()` stack allocation to avoid having to commit heap space. However, glibc then uses the aforementioned initialization process on the static portion of the TLS, which immediately faults all its pages and zeroes memory that the kernel had already cleared. In the case of *libgotcha*, the performance impact is particularly severe.

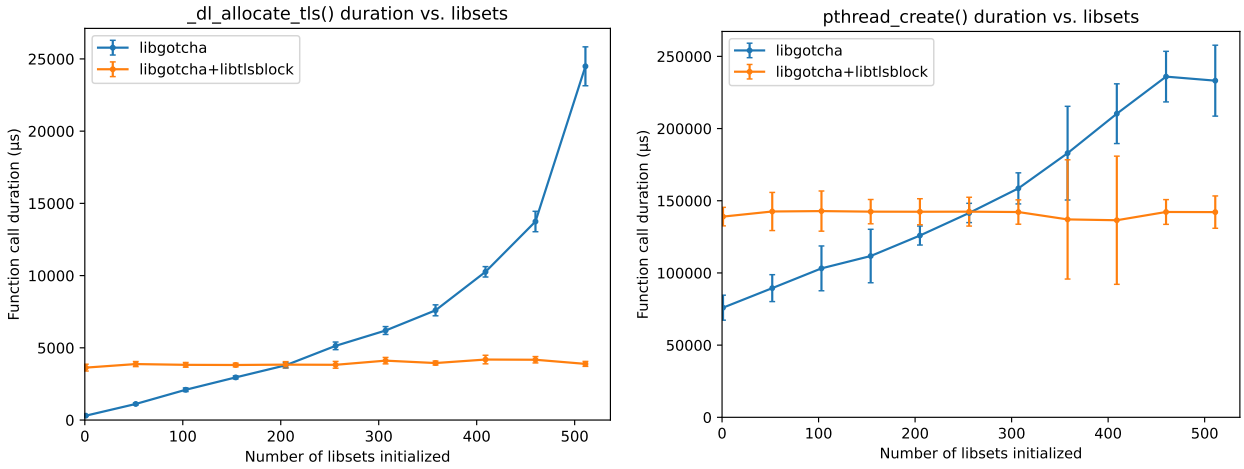
We hypothesized that this problem could be solved by initializing the entire TLS at once with a single copy-on-write mapping. To confirm this, we built a small proof-of-concept library called *libtlsblock* that interposes the `_dl_allocate_tls()` and `pthread_create()` functions with wrapper implementations that do exactly that. Our finding, shown in Figure 3.9, was that this approach incurs a latency overhead of just under 100% for thread spawns at small numbers of libsets, but that the cost remains flat as the number of libsets increases, with a break-even point at about 250 libsets. It would be easy to tweak *libtlsblock* to only activate at this number of libsets.




**(a)** Deno JavaScript runtime

**(b)** "Dyno" JavaScript runtime

**Figure 3.8:** Effect of *libgotcha* on process startup




**(a)** `_dl_allocate_tls()` in isolation

**(b)** `pthread_create()` and `pthread_join()`

**Figure 3.9:** Effect of *libgotcha* on thread spawn latency, with and without *libtlsblock*

> “ ‘Ah! This is obviously some strange use of the word *safe* that I wasn’t previously aware of.’ ”
> — Douglas Adams, *The Hitchhiker’s Guide to the Galaxy*

# Chapter 4

# Rethinking POSIX safety: *libas-safe* and *libac-safe*

In this chapter, we explore how the isolation provided by selective relinking can be applied to make otherwise unsound Unix programs safe. Along the way, we present broken sample programs that can be automatically fixed with little or no programmer intervention, and discuss practical applications to real-world systems. Through its presentation of two simple control libraries, this chapter also serves as a straightforward example of how to use *libgotcha*. The insights from this chapter, both about selective relinking and about POSIX safety itself, will be useful when we later explain the workings of *libinger*, a more complicated control library.

The reader will probably not be surprised to learn that the POSIX specification guarantees that most C library functions are thread safe; that is, assuming they are not explicitly instructed to use the same resources, it is safe to call them from concurrent kernel threads within the same process. However, POSIX also defines two less familiar types of safety that become relevant when an individual thread does something other than execute a single task to synchronous completion or failure:

**Async-signal safety.** Async-signal-safe functions are those library functions that can be safely called from a signal handler that has interrupted a non–async-signal-safe function (or conversely, can be safely interrupted by a signal handler that calls non–async-signal-safe functions).

**Async-cancel safety.** Async-cancel-safe functions are those library functions that can be called by asynchronously-cancelable threads, which we first saw in Section 2.1. One can ostensibly cancel such threads at any point in their execution; however, POSIX marks almost no functions as async-cancel safe, so in practice the feature is only useful for threads executing a compute-bound loop with no I/O or other reliance on the OS.

Along with thread safety, these two classes of safety exist because of nonreentrant interfaces (functions whose signatures do not expose all of the shared state they use). To understand the need for these two classes of safety, it is helpful to consider how one might implement a thread-safe function. For the sake of this discussion, consider the nonreentrant pseudorandom-number generator `rand()`, which takes no arguments but returns a random number. Clearly, the function needs some kind of entropy pool to produce such a number, and since the caller doesn’t provide it with any information, the `rand()` function must manage the entropy pool itself. This implies

the function has internal state that is implicitly shared among all its callers. The simplest way to implement an entropy pool is to feed some seed value to a one-way function, and use the resulting pseudorandom number as the new seed value. Thus, the entropy pool needs only to store the current seed value.

It's easy to see that a trivial implementation of `rand()` using the aforementioned approach is subject to data races when used by multiple threads: the function must update the entropy pool on each invocation, and concurrent accesses may interleave. Establishing thread safety is as easy as using a thread-local variable to maintain a separate entropy pool for each thread of execution, thereby eliminating the shared state. Unfortunately, this mitigation is applicable neither to async-signal safety nor to async-cancel safety: In the former case, there is no analogue of thread-local variables capable of retargeting data accesses depending on whether a signal handler is running. In the latter case, a function that mutates state that must be shared *between* threads is likely to corrupt such state if cancelled in the middle of writing to it, even if the function employs concurrency control to prevent data races. (One such example is the `malloc()` family of dynamic memory allocation functions, which carve their allocations out of a fixed heap. Although they take a lock on a portion of the heap while reserving each allocation, cancelling them during this critical section will result in the lock never being released and the affected portion of the heap becoming unusable.)

## 4.1 Establishing async-signal safety: *libas-safe*

Our approach to automatically establishing async-signal safety is to repurpose selective relinking to isolate signal handlers from the rest of the program. We do so by running the entire program, with the exception of uninterruptible library functions (Section 3.6) and custom signal handlers, in a newly-allocated libset. When handling a signal, we switch to the starting libset before executing its handler, then switch back before returning to the rest of the program.

As a demonstration of our technique, we have implemented *libas-safe*, a tiny runtime comprising 127 lines of C code that automatically fixes programs whose signal handlers call functions that are not async-signal safe. To use it, you either preload it at load time or link your buggy application directly against it at build time. Note that it only fixes bugs truly arising from async-signal safety: it will neutralize most resulting undefined behavior, but it will not address logic errors in the program itself (e.g., a handler's attempt to traverse a corrupt or otherwise inconsistent data structure). Furthermore, it is a proof of concept and there are cases it does not bother to handle. Most notably, it does not isolate handlers for different signals from one another, so programs that handle multiple signals must ensure the other(s) are masked while any handler that calls unsafe functions is running.[1]

To avoid affecting the initialization of the C runtime, *libas-safe* performs its own setup as late as possible by replacing libc's `__libc_start_main()` function, responsible for calling the program's `main()` function. Because it is an internal control library (Section 3.7), doing so is as simple as defining a non-static function with that name, which automatically becomes a forced interposition (Section 3.7.1). Our replacement wraps the libc implementation, but allocates and

[1]Technically, this stipulation is slightly stronger than necessary, both in terms of scope (all other handlers) and enforcement mechanism (signal masks). The exact requirement is that no two handlers that both use unsafe functions can be allowed to interleave their execution.

switches to a new libset just before jumping to `main()`. It also registers an uninterruptible return callback (Section 3.6.2) and checks an environment variable to determine wither to run in verbose mode and log its actions.

The bulk of *libas-safe*'s code merely wraps the `sigaction()` function for installing signal handlers. We show the replacement for this function in Listing 4.1, slightly simplified for brevity.[2] If used to set a signal's disposition to default (`SIG_DFL`) or ignored (`SIG_IGN`) or query the configuration of a signal without a custom handler, neither of the conditionals is taken and it defers to the underlying `sigaction()` (in this case, *libgotcha*'s own wrapper). Otherwise, if the caller is installing a custom handler, it saves a pointer to the provided handler into the persistent `handlers` array and installs its own `stub()` function as the handler instead; this function expects three arguments rather than System V's traditional one, so it sets the `SA_SIGINFO` flag [61]. If the caller is querying the configuration of a custom handler, it looks up the handler the program had requested and furnishes that instead of a pointer to `stub()`.

The `stub()` function serves as a wrapper for each installed signal handler, and is shown in Listing 4.2. If the program is interruptible (that is, the next libset is not equal to the starting one), neither of the conditionals is taken. In this case, the wrapper simply switches to the starting libset, calls the real handler for the arriving signal, then resets the libset.

Things are more complicated if the signal arrives while uninterruptible code is running, in which case *libas-safe* defers invoking its handler until the end of the uninterruptible section. In this case, `stub()` takes its `else if` branch and stores the `siginfo_t` structure describing the cause of the signal into the thread-local `pending` variable.[3] It then changes the signal mask of the *calling* code to block the signal from arriving and returns without invoking the handler. Whenever the program becomes interruptible again, *libgotcha* will invoke the `restorer()` callback, also shown in Listing 4.2.

If it finds a deferred signal to deliver, the callback sends the current thread that signal if it is not already pending (i.e., if it has not arrived again since the instance that prompted us to defer it). It then uses `sigsuspend()` to temporarily unblock the signal and atomically wait for its handler to run. This jumps back to `stub()`, which now enters its `if` branch, sets the signal as no longer deferred, configures it to be unblocked upon return from the handler, and substitutes the saved `siginfo_t` for the real one (in case `restorer()` had to signal the thread). Finally, it calls the real handler and leaves the starting libset.[4] (Note that deferring a signal only works assuming the handler does not provide a service that is necessary to continue execution. For instance, it is nonsensical to defer handlers that grow exhausted memory allocations or resolve faulting addresses, such as *libgotcha*'s own signal handler from Section 3.5.2. Handlers for signals such as segmentation fault that cause the architecture to resume by reexecuting the faulting instruction are more likely to exhibit this property.)

[2]Compared to our actual prototype, the version in these source listings runs internal *libas-safe* wrapper code with the same signal mask that the caller requested for *its* signal handler. This can lead to conflicts between *libas-safe*'s own handlers for different signals, or even between its handler for one signal and that handler itself if the caller installs the handler with the `SA_NODEFER` flag. Unrelatedly, the depicted version of the `sigaction()` wrapper does not roll back its changes if the underlying implementation reports an error.

[3]Our prototype does not support deferring more than one distinct signal at a time, and always forwards the `siginfo_t` corresponding to the first instance thereof to arrive. It follows the semantics of non-realtime Unix signals and only delivers a deferred signal once, regardless of how many times it occurred while blocked.

[4]If curious why restoring the libset in this way works, see the sister footnote in Section 3.6.2.

```
typedef void (*handler_t)(int, siginfo_t *, void *);

static struct handler_t *handlers;
static bool verbose;

int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact) {
       handler_t oldact = handlers[signum];
       struct sigaction sa;
       if(act && act->sa_sigaction != SIG_DFL
                     && act->sa_sigaction != SIG_IGN) {
              // We have been asked to install a custom signal handler.
              if(verbose) fprintf(
                     stderr,
                     "LIBAS-SAFE:␣sigaction()␣installing␣signal␣%d␣handler\n",
                     signum);

              memcpy(&sa, act, sizeof sa);
              handlers[signum] = sa.sa_sigaction;
              sa.sa_sigaction = stub;
              sa.sa_flags |= SA_SIGINFO;
              act = &sa;
       }

       // Call the real sigaction().
       int status = libgotcha_sigaction(signum, act, oldact);
       if(oldact && oldact->sa_sigaction == stub)
              // We have been asked to provide the previous configuration.
              // Fib that we installed the provided handler, not our wrapper.
              oldact->sa_sigaction = oldact;
       return status;
}
```

**Listing 4.1:** *libas-safe*'s `sigaction()` replacement

```
static thread_local siginfo_t pending;

static void stub(int no, siginfo_t *si, ucontext_t *uc) {
	libset_t libset = libset_thread_get_next();
	if(pending.si_signo) {
		// It is time to deliver a signal we had previously deferred.
		assert(pending.si_signo == no);
		pending.si_signo = 0;
		sigdelset(&uc->uc_sigmask, no);
		si = &pending;
	} else if(libset == LIBGOTCHA_LIBSET_STARTING) {
		// The program is uninterruptible; we need to defer delivery.
		if(verbose) fprintf(
			stderr,
			"LIBAS-SAFE:␣stub()␣deferring␣handling␣of␣signal␣%d\n",
			no);

		memcpy(&pending, si, sizeof pending);
		libgotcha_sigaddset(&uc->uc_sigmask, no);

		// Do not call the handler at this time.
		return;
	}

	libset_thread_set_next(LIBGOTCHA_LIBSET_STARTING);
	handlers[no](no, si, co);
	libset_thread_set_next(libset);
}

static void restorer(void) {
	if(pending.si_signo) {
		// There is a deferred signal to deliver.
		sigset_t ready;
		sigpending(&ready);
		if(!sigismember(&ready, pending.si_signo))
			libgotcha_pthread_kill(pthread_self(),
				pending.si_signo);

		sigset_t full;
		libgotcha_sigfillset(&full);
		sigdelset(&full, pending.si_signo);
		sigsuspend(&full);
	}
}
```

**Listing 4.2:** *libas-safe*'s signal handler wrapper and control library callback

While the technique employed by *libas-safe* makes it easier to write signal handlers in C, this is not its most exciting application. Async-signal safety is a very specific case that otherwise sound systems programming type systems have long struggled to handle. Rust is no exception, and writing signal handlers requires `unsafe` code. Yet this style of runtime assistance may offer a route to lifting this requirement in situations where the load-time and runtime costs are acceptable.

### 4.1.1 Automatically repaired example program

Listing 4.3 shows a small example program whose signal handler illegally calls the `printf()` function, which is not async-signal safe. It configures a custom signal handler and an interval timer that invokes it every 10 µs, then prints a circular spinner to visually indicate that the program is making progress. When run on glibc 2.29, the program reliably deadlocks in under 10 s.

The deadlock occurs because `main()`'s call to `fflush()` and `handler()`'s call to `printf()` both take a lock on the line-buffered `stdout` stream. Eventually, the signal arrives during the critical section within `fflush()`, causing the thread to block on the same lock it already holds. It is worth emphasizing that this is undefined behavior that is not guaranteed to cause any problem. As such, a change introduced in a subsequent glibc release replaced this particular lock with a "recursive" one that can be locked multiple times by the same thread without blocking. Although this hides the issue on at least glibc 2.33, it does not fix the program, which remains vulnerable to future platform changes.

Running the program with the environment variable `LD_PRELOAD` set to `./libas-safe.so` fixes the deadlock automatically. It does so by resolving the standard library calls and `stdout` reference to a different copy depending on whether they occur inside or outside the signal handler. In this way, the stream uses two separate buffers guarded by separate locks. Note that this does alter the stream's interleaving behavior; however, as the upstream glibc change demonstrates, these semantics were undefined to begin with.

The program also serves as a demonstration of a second way to fix the problem: instead of weakening the protection against interleaving writes to the terminal, we can prevent the interleaving function calls from happening in the first place. The *libgotcha* implementation recognizes a configuration environment variable that can be used to treat the entirety of `libc.so` (as opposed to only allowlisted functions) as uninterruptible code.[5] Enabling both this and *libas-safe*'s verbose mode shows that deferring signal arrival also fixes the program's deadlock. Here is an example invocation:

```
$ LD_PRELOAD=./libas-safe.so LIBGOTCHA_SHAREDLIBC= LIBAS_VERBOSE= ./signal
libgotcha notice: Treating entirety of libc as shared code
LIBAS-SAFE: as_safe() initializating...
LIBAS-SAFE: sigaction() installing signal 14 handler
LIBAS-SAFE: stub() deferring handling of signal 14
In signal handler
In signal handler
```

[5] We added this feature when we were finalizing the allowlist, as an easy way to eliminate an incomplete allowlist as a possible cause of program misbehavior. Note that it is intended only for debugging, as it violates *libgotcha*'s design assumption that all writeable global variables are private to each interruptible region (Section 3.6).

```
static void handler(int ignored) {
        printf("In␣signal␣handler\n");
}

int main(void) {
        struct sigaction sa = {
                .sa_handler = handler,
        };
        sigaction(SIGALRM, &sa, NULL);

        struct timeval tv = {
                .tv_usec = 10000,
        };
        struct itimerval it = {
                .it_interval = tv,
                .it_value = tv,
        };
        setitimer(ITIMER_REAL, &it, NULL);

        char spinner = '|';
        while(true) {
                printf("%c\b", spinner);
                fflush(stdout);
                switch(spinner) {
                case '|':
                        spinner = '/';
                        break;
                case '/':
                        spinner = '-';
                        break;
                case '-':
                        spinner = '\\';
                        break;
                case '\\':
                        spinner = '|';
                        break;
                }
        }
        return 0;
}
```

**Listing 4.3:** Example program with a signal handler that causes undefined behavior

```
LIBAS-SAFE: stub() deferring handling of signal 14
In signal handler
In signal handler
In signal handler
LIBAS-SAFE: stub() deferring handling of signal 14
In signal handler
In signal handler
LIBAS-SAFE: stub() deferring handling of signal 14
...
```

The attentive reader may notice that the signal handler has a second mistake: it does not save and restore the `errno` variable, thereby changing its value in the middle of execution and potentially interfering with error detection or recovery outside the handler. Interestingly, *libas-safe* makes this step unnecessary by providing a separate copy of `errno` (along with the rest of libc) for the code inside versus outside signal handlers. (Unless, that is, *libgotcha* is operating in the special uninterruptible libc execution mode we just saw.)

While this example may seem contrived, it is easy to accidentally introduce this class of bug into a large application simply by calling a helper function from a signal handler without first examining its implementation (and that of every function *it* might call) to verify async-signal safety. Such issues have been used to conduct arbitrary code execution attacks on widely-deployed software, including gaining root access on systems with vulnerable installations of the (setuid `root`) Sendmail and GNU Screen servers [74].

## 4.2 Establishing async-cancel safety: *libac-safe*

To establish async-cancel safety, we apply selective relinking in a slightly different manner. Instead of establishing partial memory isolation between signal handlers and the rest of the program, we establish it between each thread and every other. Whenever the program spawns a new thread, we allocate and install a private libset for it, then enable POSIX asynchronous cancelability. In this way, every kernel thread in the application except for the main one becomes cancelable at almost any point in its execution.

Our demo of this approach is called *libac-safe*, and consists of 119 lines of C that makes threads asynchronously cancelable. As with *libas-safe*, it can be either preloaded or linked against like any other library. And as before, it only fixes problems arising in library functions due to async-cancel safety, not program logic errors. In addition to observing traditional concurrency control practices, the developer of an application using asynchronously cancelable threads must exercise extreme caution when interacting with threads that might be cancelled in this way or accessing any data structures that such threads modify. Our prototype is experimental and omits important features such as automatic cleanup of resources allocated by cancelled threads (a topic we will revisit in Chapters 5 and 6).

As an internal control library, *libac-safe* works by defining a forced interposition function (Section 3.7.1) that wraps `pthread_create()`. As shown in Listing 4.4, the first time the application creates a thread, the library bootstraps itself by registering uninterruptible call and return callbacks with *libgotcha* (Section 3.6.2). The former callback is invoked any time the thread calls an uninterruptible function (Section 3.6) and automatically transitions the thread back to the de-

fault cooperative cancellation mode, wherein certain calls into the C standard library implicitly check whether there is an outstanding request for the thread to be cancelled. The latter callback happens at the end of the uninterruptible region, and transitions the thread into preemptive cancellation mode once more [51]. Once the library has been bootstrapped or on subsequent calls, the wrapper simply packages up the supplied pointer to the thread's main function and arguments along with a newly-assigned libset. It then calls into the real `pthread_create()`, substituting the *libac-safe* main function `wrapper()` and arranging for it to be passed all of these items.

Listing 4.5 shows the additional initialization and teardown we perform on each thread. Once libpthread has created the new kernel thread, it runs our `wrapper()` on it, which configures libpthread to run the `release()` handler if the thread ever gets cancelled. It then sets the next libset to the one allocated for this thread and initializes the locale selections of its copy of libc.[6] Then it calls into the thread's main function. Assuming the thread does not get cancelled, this will eventually return back, at which point `wrapper()` will return to the starting libset, set a flag to indicate that the thread ran to completion, and call the `release()` handler. This will skip the first conditional in the latter function, save the libset identifier for future zero-cost reuse by the parent thread (if it does not already have one saved), and deallocate our packaged thread information.

If instead the thread gets prematurely cancelled, `release()` will be called and find that the thread is not tagged as `finished`, at which point it will leave the thread's private libset then forcefully reinitialize it. Thereafter, it proceeds as before. Recall that cancellation will have happened cooperatively if the thread was executing uninterruptible code and preemptively otherwise.[7] This respects the selective relinking safety model; that is, assuming the allowlist is configured correctly and glibc's implementation of cooperative cancellation is sound, cancellation will neither corrupt the starting libset nor create dependencies between it and any thread's private libset (Section 3.6).

Notice that, regardless of what is causing the thread to terminate, it attempts to pass its now-unused libset's identifier back to its parent for reuse on the next spawn. The decision to reuse happens in `pthread_create()`'s call to the `alloc_libset()` helper function, also shown in Listing 4.5. This version takes two related implementation shortcuts, neither of which is central to our approach's design: (1) Terminating child threads assume their parent still exists and directly read and write its thread-local `reuse` record. (2) Each parent thread only remembers up to one reusable libset,[8] so if two or more of its child threads exit in between any two consecutive times that it spawns, it will leak a libset that the process will never reuse. The correct solution is to use a pool allocator for libsets, and we will see an example of this in Chapter 5.

In summary, *libac-safe* makes it possible for Unix applications to use asynchronous thread cancellation. Thus, developers can now leverage a feature that, as we saw in Section 2.1, was

[6] Without this thread-specific initialization, calls to `ctype.h` functions would invoke `NULL` pointers inside glibc. The reason we have to invoke it manually is that libpthread does so from within its `pthread_create()` implementation, but that (intentionally) happens before we switch away from the starting libset. Third-party libraries do not receive such special glibc initialization treatment; instead, their ELF constructors handle setup. Technically, a direct call into `__ctype_init()` as shown in the listing will not work because *libac-safe*'s status as an internal control library means that its calls only ever resolve back into the starting libset (Section 3.7). Instead of using a simple function call, it uses an additional *libgotcha* API to look up the specific function pointer for the allocated libset in the shadow GOTs, then makes an indirect call.

[7] The current implementation unnecessarily reinitializes the libset following cooperative cancellation. One could avoid this by setting the finished flag in the `cooperative()` callback, then unsetting it again in `preemptive()`.

[8] Because child threads do not atomically update the parent's variable, the implementation does not guarantee *which* libset will be reused in case of a tie. This is still safe because all contending libsets are valid for reuse.

```
static thread_local libset_t reuse;

struct wrapper {
        void *(*fun)(void *);
        void *arg;
        bool finished;
        libset_t libset;
        struct reuse *parent;
};

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*fun)(void *),
                   void *arg) {
        static bool bootstrapped;
        if(!bootstrapped) {
                libset_register_callback(cooperative);
                libset_register_returnback(preemptive);
                bootstrapped = true;
        }

        struct wrapper *args = malloc(sizeof *args);
        args->fun = fun;
        args->arg = arg;
        args->finished = false;
        args->libset = alloc_libset();
        args->parent = &reuse;
        return pthread_create(thread, attr, wrapper, args);
}

static void cooperative(void) {
        pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
}

static void preemptive(void) {
        pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
}
```

**Listing 4.4:** *libac-safe*'s `pthread_create()` replacement and control library callbacks

```
static void *wrapper(wrapper *wrapper) {
        pthread_cleanup_push(release, wrapper);

        libset_thread_set_next(wrapper->libset);
        __ctype_init();

        void *res = wrapper->fun(wrapper->arg);

        libset_thread_set_next(LIBGOTCHA_LIBSET_STARTING);

        wrapper->finished = true;
        pthread_cleanup_pop(true);
        return res;
}

static void release(void *wrapper) {
        if(!wrapper->finished) {
                libset_thread_set_next(LIBGOTCHA_LIBSET_STARTING);
                libset_reinit(wrapper->libset);
        }
        if(*wrapper->parent != LIBGOTCHA_LIBSET_STARTING)
                // Our parent thread does not already have a libset to reuse.
                *wrapper->parent = wrapper->libset;
        free(wrapper);
}

static libset_t alloc_libset(void) {
        if(reuse != LIBGOTCHA_LIBSET_STARTING)
                // This thread has a leftover libset to reuse.
                return reuse;
        else
                // We have to use a fresh one.
                return libset_new();
}
```

**Listing 4.5:** *libac-safe*'s thread initializer and cleanup handler

practically useless out of the box. The library also makes a good case study of a *libgotcha* control library for which it is correct to adopt the default behavior of tying TCBs to kernel threads (Section 3.7.2), since the kernel thread corresponds to the unit of cancellation, and therefore memory isolation as well. Because of this, *libgotcha* does not need to reinitialize TLS areas, as they are specific to each thread and not reused with the libset.

### 4.2.1 Program repaired with the help of our system

Listing 4.6 shows an only slightly contrived example program that tries to use asynchronous thread cancellation. It attempts to simulate performing a large number of cancelable DNS lookups: The main thread starts by performing a reverse lookup on a link-local IPv4 address (in order to populate a socket address structure). It then enters an infinite loop, each iteration of which updates a circular spinner to indicate that the program is making progress,[9] spawns a thread to resolve the hostname back to an address, cancels the thread, and joins on it before spawning the next. Each time a thread terminates prematurely, the loop increments a success counter; each time one runs to completion, it prints out the current counter and the hostname the thread was passed. Upon creation, each thread immediately sets itself as asynchronously cancelable, performs a forward DNS lookup, and exits.

The program does not work. Asynchronous cancellation both fails to interrupt some of the threads (likely due to the proximity between the call and a library function that performs system calls) and eventually deadlocks the program, as indicated by the spinner becoming frozen. Here is the output of a representative run, which lasted under three seconds before hanging:

```
69437 localhost
75670 localhost
128996 localhost
-
```

Running the program with the environment variable `LD_PRELOAD` set to `./libac-safe.so` almost fixes the problem with no programmer intervention. All of the threads are cancelled before they can complete, and the only problem is that each of the calls to `getnameinfo()` allocates a new file descriptor that is leaked upon the thread's death. Eventually, this results in the program crashing because it has too many open file descriptors.

After adding a cleanup handler, the program runs apparently forever without deadlocking, allowing any of the threads to finish, or leaking file descriptors. The cleanup handler is as follows, and is registered from `thread()` using the `pthread_cleanup_push()` function as in Listing 4.5:

```
static void handler(void *ignored) {
{
      close(STDERR_FILENO + 1);
}
```

[9] We implement this as in Section 4.1.1.

```
#define NAMESZ 10

struct args {
	socklen_t addrlen;
	const struct sockaddr *addr;
	char name[NAMESZ];
};

static void *thread(struct args *args) {
	pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
	getnameinfo(args->addr, args->addrlen, args->name, NAMESZ, NULL, 0, 0);
	return NULL;
}

int main(void) {
	struct addrinfo *ai;
	getaddrinfo("127.0.0.1", NULL, NULL, &ai);

	struct sockaddr_storage sa;
	socklen_t salen = ai->ai_addrlen;
	memcpy(&sa, ai->ai_addr, salen);
	freeaddrinfo(ai);
	ai = NULL;

	struct args args = {
		.addrlen = salen,
		.addr = (struct sockaddr *) &sa,
	};
	char spinner = '|';
	unsigned cancelled = 0;
	while(true) {
		print_and_advance_spinner(&spinner);

		pthread_t tid;
		void *res;
		pthread_create(&tid, NULL, thread, &args);
		pthread_cancel(tid);
		pthread_join(tid, &res);
		if(res != PTHREAD_CANCELED)
			printf("%d␣%s\n", cancelled, args.name);
		else
			++cancelled;
	}
	return 0;
}
```

**Listing 4.6:** Example program using asynchronous thread cancellation

> “ A half-read book is a half-finished love affair. ”
> — David Mitchell, *Cloud Atlas*

# Chapter 5

# Function calls with timeouts, revisited: the *libinger* library

In Chapter 2, we introduced and motivated lightweight preemptible functions, a novel concurrency abstraction pairing synchronous invocation with preemption. At that time, we covered the design principles underlying the API for managing preemptible functions; in this chapter, we discuss the API itself in more detail and the design and implementation of *libinger*, the library that provides preemptible functions.

We start by giving the full *libinger* C interface in Listing 5.1. The `launch()` and `resume()` functions work as already described: the former creates a new preemptible function and lets it run on the caller's thread for the specified number of microseconds (which may be zero), and the latter resumes a preemptible function that had become paused after exhausting its time budget. The new `cancel()` function allows the caller to discontinue a paused preemptible function rather than allowing it to run to completion. Finally, the `pause()` function may be called from within a preemptible function to cooperatively yield. It immediately pauses the function and returns to its caller, just as if the function had been preempted.

The Rust interface appears in Listing 5.2 and differs in several important ways:

```
struct linger_t {
        bool is_complete;
        cont_t continuation;
};

typedef void (*Function)(void *);

linger_t launch(Function func, uint64_t time_us, void *args);
void resume(linger_t *cont, uint64_t time_us);
void cancel(linger_t *cont);
void pause(void);
```

**Listing 5.1:** Preemptible functions extended C interface

```
// Tagged union
pub enum Linger<T> {
        Completion(T),
        Continuation(Continuation),
        Poison,
}

pub fn launch<T: Send>(func: impl FnOnce() -> T + Send,
                       time_us: u64) -> Result<Linger<T>>;
pub fn resume<T>(func: &mut Linger<T>,
                 time_us: u64) -> Result<&mut Linger<T>>;
pub fn pause();

impl<T> Linger<T> {
        pub fn yielded(&self) -> bool;
}

// Destructor
impl Drop for Continuation { ... }
```

**Listing 5.2:** Preemptible functions Rust interface

**Closure support.** We leverage Rust's first-class closures to enable the caller to pass to `launch()` a function that captures state from its environment. We expect the caller to provide any inputs to the function in this manner. Unlike with the C interface, there is no need to wrap the arguments in a struct when there are multiple, or to pass an empty value when there are none.

**Type safety.** The aforementioned interface change means that the Rust wrapper functions do not erase the types of the preemptible function's parameters by diluting them to a `void *`; thus, the preemptible function does not have to perform an unsafe cast before using them, and the compiler can still type check the program. Furthermore, both `launch()` and `resume()` are generic on the preemptible function's return type: instead of a `linger_t`, they return a tagged union. Once the preemptible function runs to completion, the caller may destructure this type to retrieve the function's return value. Because the union is tagged, it is impossible to destructure it to retrieve the return value unless the function has truly run to completion.

**RAII.** Our Rust interface adheres to the RAII (Resource Allocation Is Initialization) idiom, allowing continuation deallocation to happen automatically. Unlike the C interface, the Rust one has no `cancel()` function; instead, its continuation objects implement the language's `Drop` trait. Whenever a continuation goes out of scope without being consumed by running to completion, the language calls its destructor, which implicitly performs a cancellation.

**Safe concurrency.** The `launch()` function requires that the preemptible function closure implement the language's `Send` trait, which is true provided that all of the values it captures have

this trait. In Rust, a type is `Send` if and only if ownership of it can be safely transferred between threads. This includes all objects that do not contain any references, as well as those that do but only to data that is safe to access concurrently (`Sync` in Rust parlance) [55]. This restriction on the closure means that any attempt to share state between a caller and its preemptible function without the use of appropriate concurrency control will fail at compile time.

**Flexibility.** The requirement that Rust preemptible functions be `Send` is similar to the restrictions imposed by the standard library's thread `spawn()` interface. However, `launch()` differs in an important way: unlike a thread, a preemptible function is not restricted to the `'static` lifetime, and so is able to accept references to local variables and other dynamically-allocated data. In contrast, attempts to transfer such references to a thread would result in a compile error. What makes it safe to use all lifetimes with preemptible functions is the fact that they execute synchronously, and therefore cannot outlive the calling context without first becoming paused. If a preemptible function times out, the Rust compiler knows the lifetime of any references it has captured, so any attempt to pass the paused closure to a scope where its shared data no longer exists will be met with a compile error.

**Composability.** In addition to the closure being `Send`, the opaque `Continuation` type is as well. This means that a preemptible function can be launched on one thread, become paused, then be moved to another thread and resumed there. Applications may use this trick to move long-running tasks off the critical path, but it is especially important for implementing thread libraries. In fact, as we will see in Chapter 7, it makes it easy to implement preemptive thread libraries in userland.

## 5.1 Shared responsibility for concurrency control

The preceding points about concurrency restrictions in the Rust API may seem incongruous with Chapter 3, which spent dozens of pages introducing selective relinking, a technique billed as solving the concurrency perils of preemptible functions. In fact, that runtime exists to address a completely separate (but equally critical) problem.

Adding preemptible functions to an application actually introduces two distinct forms of concurrency. Both stem from the fact that code within the same thread is now allowed to interleave its execution at almost entirely arbitrary points, but they differ in whether the code in question can possibly anticipate this problem, and therefore have any hope of addressing it.

First, there is concurrency involving the libraries that the program depends on, many of which were probably authored by third parties. Although such libraries now execute concurrently with preemptible functions by virtue of being used from the same kernel thread, they conceptually "predate" preemptible functions' existence; that is, they cannot even be expected to be aware of this concurrency. The job of *libgotcha* is to reconcile this, first by establishing an isolation boundary between each library (and the executable, for that matter) and the rest of the program, and then by deferring preemption where shared state is still unavoidable.

Separately, there is concurrency involving the code that uses the preemptible functions abstraction (i.e., implements and invokes preemptible functions). Not only *can* this code take measures to ensure this concurrency is safe: it has to be the one to do so. This is because there is often

a legitimate need to explicitly exchange information between a preemptible function and the surrounding program, in which case the possibility of interleaving must be directly confronted. As a simple example, consider a preemptible function that is populating some data structure for later use by the rest of the program. Imagine that the function exhausts its time budget and the application is running behind schedule and opts to cancel it. Should the application need to retrieve the work done so far, it must use its knowledge of how the preemptible function mutates the data structure to individually validate each portion thereof before trusting it to be in a consistent state. (The preemptible function can make this easier by exposing a record of its progress that indicates what parts of its data structures are consistent.)

Even without using cancellation, the traditional hazards of concurrency arise, just as they do with state-of-the-art abstractions. This problem space has long posed a challenge to systems programmers, and we do not pursue any novel solution. When using the C interface, the programmer bears complete responsibility for writing code that is free of data races. Our Rust API, however, leverages that language's first-class concurrency support so the compiler can catch such mistakes.

### 5.1.1 Locking and deadlocks

While the Rust compiler rejects all code that shares state unsafely, it is still possible to introduce correctness bugs such as deadlock [57]. This is nothing new, a common cause being an ordinary function blocking on a mutual-exclusion lock that is already held by its caller. But it is especially easy to make this mistake with preemptible functions. The developer must remember that each preemptible function runs on the same thread as its caller, so blocking is not a legitimate way for a preemptible function to synchronize except with independent threads.

Still, it is sometimes necessary for a preemptible function to protect a non-atomic resource from other code on the same thread. This is possible to do with yielding. When a preemptible function needs to acquire a mutex, it should use a try lock operation instead of a blocking lock. If it fails to acquire exclusive access, it should call `pause()` and wait for the caller to reschedule it at a later time when the resource is hopefully available. To make this more ergonomic, one could easily build a custom mutex type that used this algorithm to implement its "blocking lock" operation when called from a preemptible function.

The Rust API includes a `yielded()` method that the caller can use to determine whether a preemptible function paused cooperatively. This can be used to implement the equivalent of deadlock detection for situations where multiple preemptible functions are contending for a resource.

## 5.2 Launching a preemptible function

Invoking the `launch()` wrapper function with a nonzero time limit does the following:

1. Allocates and installs a private thread control block specific to the preemptible function (Section 5.3)

2. Captures a snapshot of the kernel thread's execution context (Section 5.4)

3. Allocates and switches to a private execution stack specific to the preemptible function (Section 5.5)
4. Allocates a preemption signal specific to the kernel thread (Section 5.6)
5. Records a timestamp shortly before invoking the preemptible function (Section 5.7)
6. Allocates and switches to a private libset specific to the preemptible function (Section 5.8)
7. Invokes the preemptible function (Section 5.9)

Several of these steps involve allocating resources assigned to each preemptible function. Some of these allocations are slow as currently implemented, but the resources are reusable once a preemptible function has completed or been cancelled. We use pool allocators to automatically reuse released resources when available. To take expensive operations off the critical path, some of the allocators preallocate a number of instances up front.

The following sections discuss each of the preemptible function invocation steps in detail.

## 5.3 Thread control blocks

As discussed in Section 3.7.2, *libgotcha* leaves it to control libraries to decide the scope of thread-local variables. However, the preemptible functions Rust API constrains *libinger*'s choice in the matter. Because the `Continuation` type is `Send`, preemptible functions may resume execution on a different thread than they were running on before becoming paused. To prevent their thread-local variables from changing out from under them, we therefore associate thread-local variables with the preemptible function instead of the thread. Application programmers should be aware that, unlike global variables, thread locals' values are not shared between a preemptible function and its defining module.[1]

To allocate a thread control block, *libinger* calls the dynamic linker's `_dl_allocate_tls()` function (the same one used by `pthread_create()`). This function creates a TCB and accompanying TLS area, but only initializes the latter. Unfortunately, there are a few TCB fields that must be initialized for the thread to operate properly, so *libinger* manually sets those.[2] It then calls `__ctype_init()` to select the correct `ctype.h` implementations for the locale. Thread control blocks are one of the resources it pool allocates and reuses unless a preemptible function is cancelled. This choice was informed by the expense of allocating TCBs as seen in Section 3.9.3; however, it is possible to instead reduce this cost using a technique like the one we prototyped in that section.

To install the TCB, we need to load it into the thread segment register using `arch_prctl()`. This is one of the functions that *libgotcha* wraps (Section 3.7.1); since *libinger* is an internal control library, it calls `libgotcha_arch_prctl()` instead to also notify *libgotcha* of the change.

[1] Users of thread pools built on top of preemptible functions need not be aware of this detail, or even of preemptible functions. This is because thread pool users must already assume their task will be scheduled on a different thread than the code that submitted it.

[2] Two of these are self-referential pointers back to the beginning of the TCB: one is for finding the TLS area (which is located just before the TCB), and one is the thread identifier returned by `pthread_self()`. There is also a pointer guard field that glibc uses to decode some of its internal pointers that are mangled as an exploit mitigation, and a field containing the kernel's thread identifier.

## 5.4 Execution contexts

The `Continuation` type inside a paused preemptible function must contain enough information to resume the function's execution from where it left off, which could be any program point because we interrupt asynchronously. To capture a snapshot of the machine registers, we use POSIX contexts, a mechanism for non-local jumps. Unlike the better known `setjmp()`/`longjmp()` interface from the C standard, POSIX contexts permit capturing an execution snapshot on one thread and resuming it on another.

One of `launch()`'s early actions is to snapshot its own execution context. It keeps this snapshot available while the preemptible function is running; if the function times out, *libinger* will use the snapshot to jump directly back into `launch()` (or `resume()`), which will populate the `Continuation` type and return it to the caller.

### 5.4.1 Using POSIX contexts safely

Any nontrivial use of the POSIX contexts interface requires some surprisingly subtle boilerplate code. Like the `setjmp()` function, the `getcontext()` function can return multiple times: it always returns once when the snapshot is initially captured, but each subsequent restoration of the context causes it to return again. Unfortunately, it provides no indication of whether it is returning for the first time, even though the caller almost always needs to know. For instance, we only want to launch the preemptible function after we capture the snapshot, not after it is restored (in which case we want to package the `Continuation` and return). A simple boolean flag does not work because the compiler is likely to store it in a register, so any updates after the capture will be undone by the restore. At a minimum, one must store the flag in a volatile variable [24].

In C, getting the flag right would be enough boilerplate to use the context, as long as we did so sensibly. However, RAII in languages such as Rust makes it very easy to implicitly create memory corruption. Consider the example in Listing 5.3, which is correct as written but deceivingly brittle. It prints:

```
Snapshot was captured!
About to restore snapshot!
Snapshot was restored!
Deallocating `closure`, `capturing`, and `snapshot`!
```

The closure here is of type `Fn(&ucontext_t) -> ()`, which means it takes a reference to a POSIX context and returns nothing. However, changing the code in the closure might affect that type. For example, imagine we had a type `OneTimePadded` for storing data that could only be unwrapped once. We might want `closure` to capture such a value and "decode" it:

```
let secret = OneTimePadded::from("About␣to␣restore␣snapshot!");
let closure = move |snapshot| {
     println!("{}", secret.into_inner());
     setcontext(snapshot);
};
```

If we make this substitution, the code still compiles. The program's output describes when the existing variables go out of scope, but how about the new `secret` variable? Recall that it

```
unsafe {
        // Closure that runs once right after we capture the snapshot
        let closure = |snapshot| {
                println!("About␣to␣restore␣snapshot!");
                setcontext(snapshot);
        };

        // Flag to track whether we are capturing or restoring the snapshot
        let mut capturing = Volatile::from(true);

        // Checkpoint the program
        let mut snapshot = MaybeUninit::uninit();
        getcontext(snapshot.as_mut_ptr());

        if capturing.read() {
                println!("Snapshot␣was␣captured!");
                capturing.write(false);
                closure(snapshot.assume_init_ref());
        } else {
                println!("Snapshot␣was␣restored!");
        }

        println!("Deallocating␣`closure`,␣`capturing`,␣and␣`snapshot`!");
}
```

**Listing 5.3:** Subtly unsound use of POSIX contexts from Rust

can only be unwrapped once; therefore the `secret.into_inner()` call consumes it (by taking ownership). When that call returns, it will go out of scope and its destructor will be invoked. This should be troubling: we said earlier that `closure` was in scope until the end of the `unsafe` block, so it would seem that `closure` would have to live that long as well (so that `secret` would still be deallocated if `closure` were never called).

In fact, the Rust compiler is aware that the closure "uses up" its captured variable. The most important implication of this is that the closure can never be called more than once, so the compiler has changed its type to `FnOnce(&ucontext_t) -> ()`. As with our imagined `into_inner()` method, calling a `FnOnce` closure consumes the closure. For this reason, the scope of `closure` has changed and it now gets deallocated either when it is called or at the end of the `unsafe` block, depending on which branch is taken. Unfortunately, when we restore the snapshot, Rust does not know that the `if` has already been taken, so it assumes it is in the latter situation and deallocates `closure` at the end, calling `secret`'s destructor in the process. If `secret` includes any heap-allocated memory, this is a double free. Note that this unsoundness was introduced not because `closure` now captures a variable, but because it *consumes* that variable.

Programmers might need to write preemptible functions that consume variables, so we want to support this case. In order for `launch()` to safely accept a `FnOnce`, however, we need to fix the potential double free. To do so, we need to make Rust "leak" `closure` in the `else` branch. We can do this by passing it to the standard library function `mem::forget()`, which takes ownership of its argument but wraps it in an untagged union so the compiler is unable to invoke its destructor. Doing so will leak any of `closure`'s locals or captured variables that are still in scope when it called `setcontext()`, so it should make sure to explicitly deallocate them before doing so by calling the standard library function `drop()`.[3] The *libinger* implementation is careful to do this with its internal variables, but the preemptible function cannot because pausing is preemptive and can occur at any point in its body (or that of most functions it might call). If it is eventually run to completion, all its variables will go out of scope normally. Alternatively, it might be cancelled before this happens (Section 5.11).

A more obvious thing that a user of POSIX contexts can do wrong is to restore a snapshot after the function that captured it has returned. This is undefined behavior because the stack may have been clobbered, taking with it any local variables and the return pointer. To avoid this and the above problems with the flag and destructors, *libinger* encapsulates boilerplate code with the solutions we have developed into a `getcontext()` wrapper function that restricts the scope of the snapshot to that of a callback function (called in the same place as `closure` from the example):

```
fn getcontext(callback: impl FnOnce(&ucontext_t) -> ())
```

Sometimes the safety of this interface is too restrictive for *libinger*'s needs. For instance, when a preemptible function times out, the continuation that `launch()` returns to its caller contains a POSIX context. If the caller ever calls `resume()` on the continuation, we must graft a new return point onto the context before restoring it to remove its dependency on the stack frame of the `launch()` call that has since returned. Restoring a context in this way is not safe in general: as we will see in the next section, it requires careful management of multiple execution stacks.

[3] The reader might notice that this will leak the closure itself. This is inconsequential unless the closure was allocated on the heap, in which case the disclaimer from Section 5.11 applies.

Rather than try to catch this class of bugs at compile time, we compile extra runtime checks into debug builds of *libinger* that validate the stacks and return point before restoring each context.

One final danger when using POSIX contexts from Rust is that the glibc implementation of the `ucontext_t` structure contains a self-referential pointer to the floating-point context. This can cause surprising undefined behavior with contexts allocated on the stack, because the Rust compiler assumes that it is safe to move objects in memory; however, this glibc implementation detail means that doing so invalidates the pointer. To work around the issue, our `setcontext()` wrapper function fixes up the self-referential pointer just before restoring the passed context. Since we implemented this part of *libinger*, Rust has introduced a `Pin` wrapper type for statically expressing that certain operations are only safe on objects that are allocated at a stable address. Instead of our approach, one could employ this feature to make such unsound calls impossible.

Ensuring that our use of POSIX contexts is free of undefined behavior has been the most challenging part of implementing *libinger*. The static and dynamic checks described in this section provide a measure of confidence (but not a full proof) that the current implementation is sound, and have also served as an indispensable tool for debugging past errors.

## 5.5 Execution stacks

When a preemptible function times out, *libinger* returns a continuation object. The caller might pass this object around the program while the function is paused, then later call `resume()` from a different stack frame. Were *libinger* not careful, this would be equivalent to restoring a POSIX context whose stack frame had already returned. To avoid this unsoundness, `launch()` allocates a dedicated execution stack and switches to it before calling the preemptible function.

Figure 5.1 shows the frames of both stacks just after we have transferred control to the user-supplied preemptible function `F()`. Note that the preemptible function can call functions normally, but that if it finishes and returns, it will reach the bottom of its stack. We need a return to transfer control back to `launch()` (or `resume()`), so we must plant a "return" address within one of those functions' code at the bottom of the preemptible function's stack. To do this, we use the `makecontext()` convenience function, which patches a POSIX context with a dedicated stack so that it restores another context when it returns. We use the snapshot we captured before switching stacks (Section 5.4) as the context to restore. This means that execution proceeds from the same point in `launch()` or `resume()` regardless of whether the function returned normally or timed out.

If the function becomes paused, we store its stack in the `Continuation` alongside the TCB so neither gets released until the program is done with the preemptible function. It is infeasible to relocate a stack in virtual memory while a preemptible function is paused, so *libinger* currently preallocates large 2-MB stacks to avoid having to resize them (as this should be large enough for any function that does not stack allocate large locals). As an implementation shortcut, it currently allocates the stacks with `malloc()` and uses a pool allocator to preallocate and reuse stacks (regardless of whether the preemptible function ended in cancellation).

Neither of these limitations is fundamental. If one wanted to avoid preallocating stacks to reduce startup time and physical memory requirements, one could allocate stacks with `mmap()` to avoid faulting the pages. This technique would also allow one to make resizeable stacks by requesting even "bigger" stacks that would expand to meet functions' needs (up to some fixed

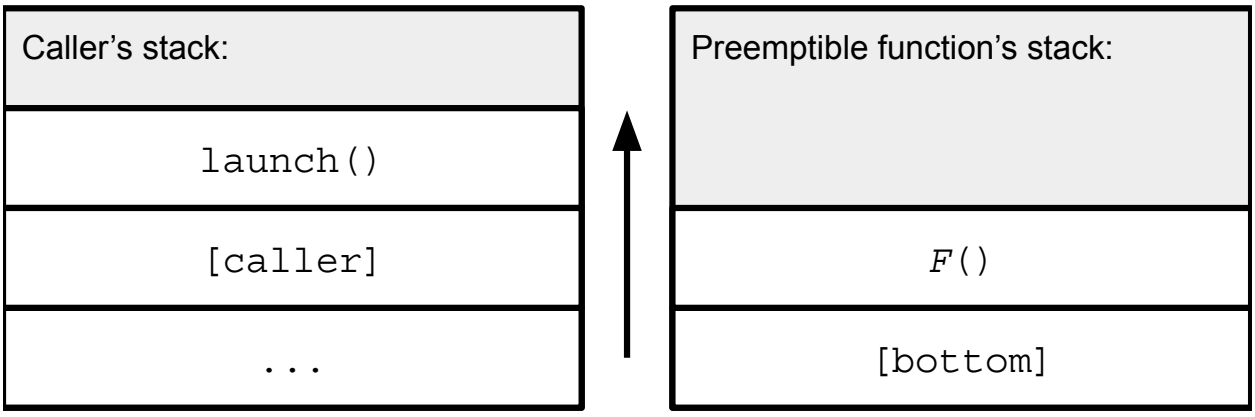



**Figure 5.1:** The stacks just after the preemptible function `F()` has been invoked

maximum size) using demand paging. If one needed truly "boundless" stack sizes, one could place an unmapped guard page at the top of each stack; if the stack tried to grow into this space, one would allocate another stack somewhere else and chain them together with another synthetic return address.

## 5.6 Signal-based preemption

The defining feature of preemptible functions is that they can be interrupted at any point. We implement this external interruption using POSIX interval timers. The `launch()` function calls `timer_create()` to request that the kernel enable fine-grained timer interrupts and periodically signal the process. When the signal arrives, control transfers to a handler function in *libinger* that may decide to pause the preemptible function or let it continue running. Unfortunately, the signal is process-directed and gets delivered to an arbitrary thread, not necessarily the one that is running the preemptible function.

To achieve thread-directed signaling, `launch()` allocates the current thread a signal number from a pool. We use the assigned signal to interrupt this specific thread. Once the thread is no longer running a preemptible function, we can release the signal for use by a different one. Of course, the signal might still be received by an unintended thread, so the first thing that our signal handler does is check whether the arriving signal is assigned to the current thread; if not, it blocks the signal on that thread. After as many time periods as the application has threads, this approach converges to delivering the signal only to its corresponding thread. Convergence is even faster when reusing a signal from the pool, as it will already have been blocked by all except one of the threads that existed when it was last used.

A limitation of this design is that the number of kernel threads running preemptible functions at any given time cannot exceed the number of different signals that the operating system pro-

vides. Linux currently has 31 standard signals, of which *libinger* uses up to 16 for preemption.[4] It currently assumes that the process will not use any of these signals for other purposes, but detecting other uses would be as simple as adding enforced interposition wrapper functions for `signal()` and `sigaction()` (Section 3.7.1).

If a signal arrives in the middle of a system call, the system call aborts and returns an error code. This is unacceptable because it would mean that moving code into a preemptible function would introduce spurious error returns from C library and POSIX functions, which the preemptible function would then have to handle. When we install our signal handler, we use `sigaction()`'s `SA_RESTART` flag to request BSD signal semantics. This hides the signal arrivals from the preemptible function by making most standard library functions transparently retry interrupted system calls. The `read()` and `write()` families of functions change their behavior in this configuration: blocking calls that have already transferred some data when the signal arrives will return early, reporting successful completion and the number of bytes processed. Programmers using these interfaces should already defend against the possibility of short counts, so this should not affect correct programs but might expose existing bugs. There are also some functions that still exit with an interruption error code; see Section 5.10 for further discussion.

### 5.6.1 Interval length and accuracy

We have described our preemption mechanism, but the question remains of how frequently the signal handler should check whether the preemptible function has timed out. For simplicity, *libinger* currently uses a single fixed scheduling quantum (timer signal interval) across all preemptible functions.

Before choosing the quantum to use, we were curious how short an interval was achievable with Linux signals on modern high-precision CPU timers. We ran an experiment to determine the effect of a POSIX timer's period on its accuracy. To measure accuracy, we wrote a small test program that installs a signal handler and configures a POSIX timer to trigger it every $T$ µs. Ideally, this handler would always be called exactly $T$ µs after its last invocation; we measured this duration and recorded the observed deviations from $T$ over 65,535 iterations. Repeating this for various values of $T$ showed that the variance is smaller than 0.5 µs for $T \geq 3$ µs, although it exhibits a warmup effect after configuring (or reconfiguring) the timer.

Of course, at a quantum of 3 µs, the CPU will be wasting much of its time on signal handling. To assess the overhead of various quanta, we wrote another test program that repeatedly computed SHA-512 sums over 64 B of data at a time. We subjected this program to `SIGALRM`s generated by a POSIX timer, varying the quantum and observing the resulting hashing throughput. Figure 5.2 shows that by a quantum of about 20 µs, throughput had reached 90% of baseline.

This study shows that it is feasible to support preemption at granularities in the tens of microseconds, up to two orders of magnitude faster than Linux's default 4-ms scheduling quantum for processes. However, it has been our experience that such small quanta pose a headache during development because they overwhelm debugging tools such as GDB and Valgrind, causing them

[4]This limit could be roughly tripled by using POSIX real-time signals, although we have not attempted this because they have slightly different behavior [62]. Alternatively, glibc on Linux offers a nonstandard `SIGEV_THREAD_ID` configuration parameter for directing timer signals at a specific kernel thread; this could be used to remove the limit (and the pool allocator) entirely [67].

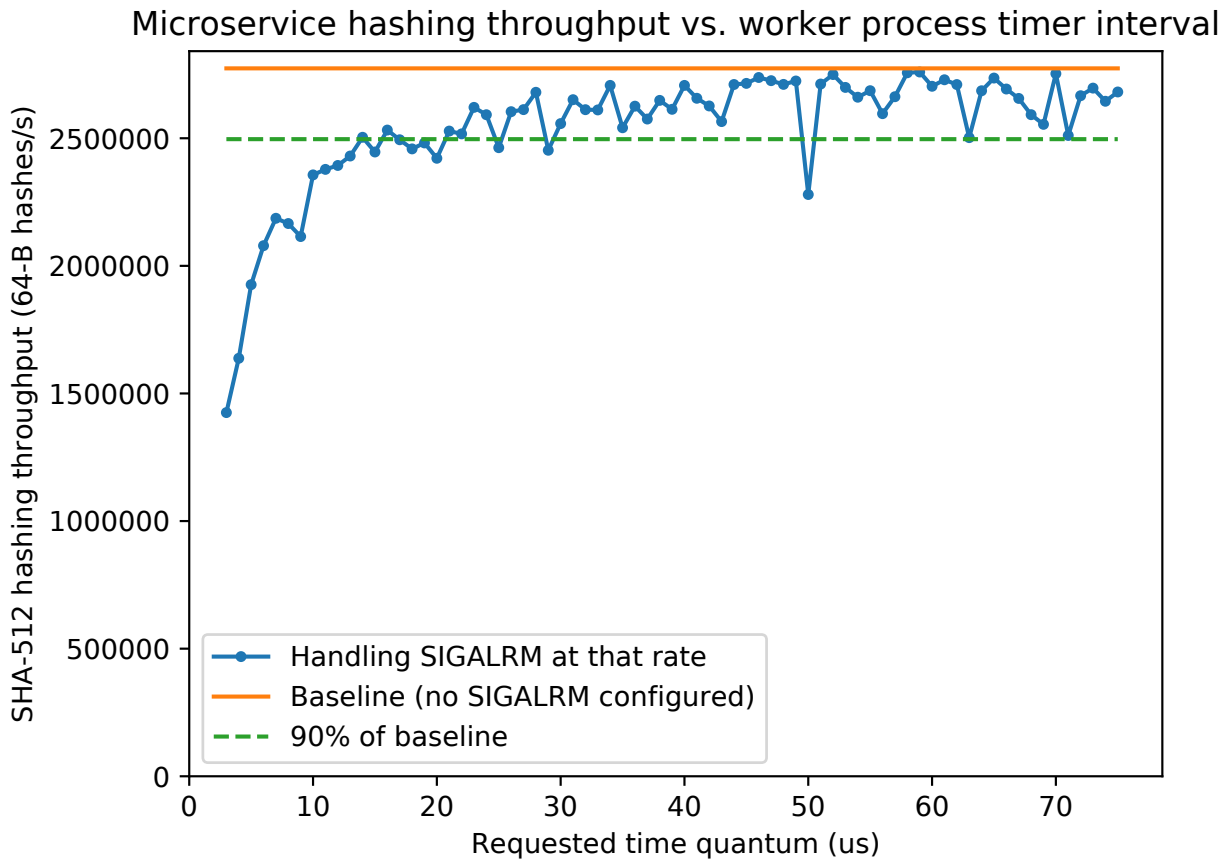



**Figure 5.2:** Effect of `SIGALRM` quantum on hashing throughput

to become unresponsive to user input (even if configured not to deliver the signal to the target program). To avoid such problems, *libinger* currently adopts a "compromise" quantum of 100 µs.

The quantum determines how small a timeout *libinger* can enforce for a preemptible function. Note that the quantum represents the duration by which a preemptible function might exceed its prescribed timeout: in the worst case, it will exhaust its time budget infinitesimally long after the handler has interrupted it, and therefore not be paused until one full quantum later (assuming the timeout is not so short that the timer is still in its warmup stage).

Competing preemption systems sometimes direct timer signals to a central "watchdog" thread that checks whether any tasks are in need of interruption and forwards a signal to the kernel thread of each that does. This avoids reducing each preemptible function's compute throughput at the expense of sacrificing one CPU core. We instead opted to use per-thread timer signals for two reasons: First, assuming most preemptible functions achieve over 90% of their baseline throughput as predicted by our benchmark, it would take more than nine cores constantly running preemptible functions to break even by committing a watchdog core. Second, although such a design may permit preemption quanta in the single-digit microseconds, the gains would be less in practice because moving the decision to pause to a watchdog core would increase the worst-case timeout overrun by the latency of propagating the signal between cores. Recent measurements place this latency at just under 5,000 cycles on Linux, which on a 2-GHz processor already represent almost 10% of the quanta achievable with marginal throughput cost. Worse, the sender thread incurs almost half of this latency [35], meaning the effect could compound on the watchdog thread and add to the overrun of other preemptible functions as well. That said, if an application could not cope with the throughput cost posed by our approach, would benefit from dropping the preemption quantum by one additional order of magnitude, or found our signal pool to constrain its scaling, it could revisit this design decision.

The use of a fixed quantum is not fundamental. One alternative option is to adjust the quantum based on the magnitude of the requested timeout, so that the preemptible function pays a compute overhead proportional to the precision of its time budget. To be effective, this may require collecting per-machine profiling data on signal timing characteristics. A separate enhancement that would benefit longer-running preemptible functions while still delivering very accurate preemption is to configure a non-repeating timer to trigger a single signal shortly before the function was scheduled to time out, then have the handler reconfigure the timer to repeat with a small quantum for the remainder of the program's run. The choice of pre-deadline duration should be informed by the machine's timer signal warmup behavior.

## 5.7 Pausing a running preemptible function

We base the decision of whether to pause a running preemptible function on elapsed wall-clock time, not actual compute time. One of `launch()`'s final actions before invoking the preemptible function is to save a timestamp. Each time our signal handler runs, it checks whether the function has exceeded its timeout. If so, it pauses it by performing an unstructured jump back to `launch()` (or `resume()`). Specifically, the location it jumps to is the snapshot captured earlier, as described in Section 5.4.

While it is possible to call `setcontext()` from a signal handler [24], there is an easier way to restore the snapshot. When installing our signal handler, we set the `SA_SIGINFO` flag to re-

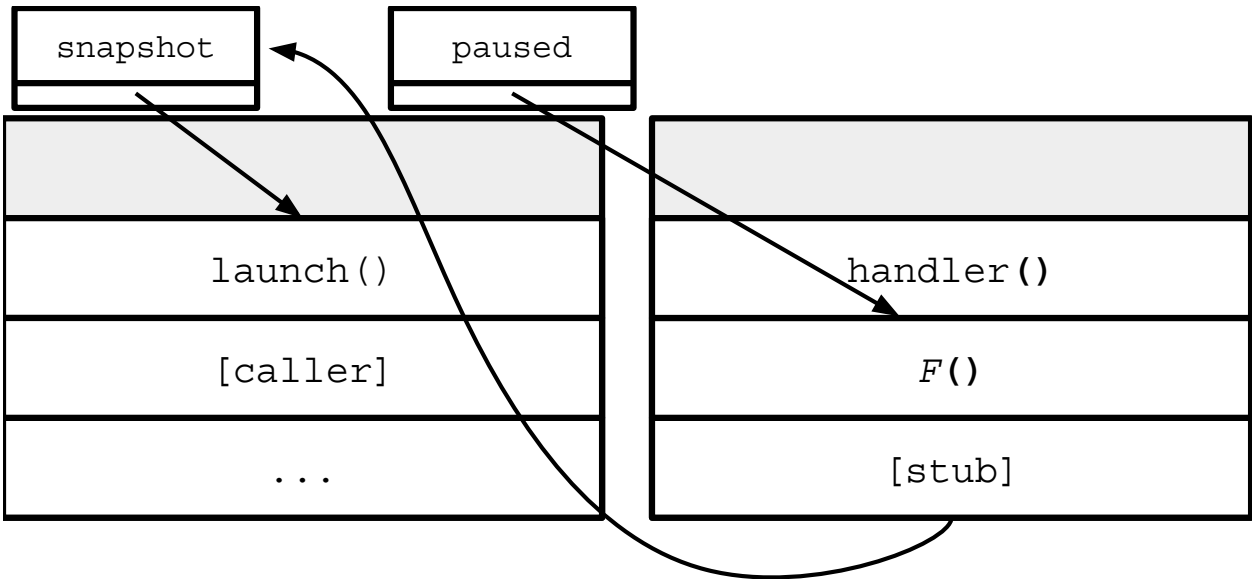



**Figure 5.3:** The stacks and continuations just after *libinger*'s signal handler has been invoked

quest that each invocation receive more context. This passes additional arguments to the handler function, one of which is a POSIX context recording the execution state just before the signal arrived. This is also the state that will be restored when the handler returns, so we implement the jump simply by overwriting it with our saved snapshot. Because the context checkpoints the registers including the stack pointer, returning also switches back to the execution stack of the preemptible function's caller.

We are only pausing the preemptible function, so the program might later want to resume it from where it left off. The preemptible function is already running on its own stack, and the context passed to the signal handler contains its remaining state. Before overwriting it, we copy its contents for `launch()` or `resume()` to package into the returned `Continuation`. Figure 5.3 shows the state of both stacks and which frames both continuations point to at the start of the signal handler's execution. Recall that the snapshot serves as the bridge between the two stacks, and represents the location that the preemptible function will return to regardless of whether the handler chooses to pause it.

### 5.7.1 Resuming a paused preemptible function

Should the application resume a paused preemptible function, `resume()` repeats many of the transition actions performed by `launch()`.[5] It reuses most resources wholesale from the `Continuation` object, but it may need to allocate a new preemption signal (if the application has transferred the paused preemptible function between kernel threads). To resume execution, it must restore the context saved by the signal handler.

Unfortunately, a restriction in the POSIX specification complicates this unstructured jump. The behavior of calling `setcontext()` on a POSIX context obtained from a signal handler is

[5] In fact, there is so much overlap that we implement `launch()` simply by creating a `Continuation`, and then calling `resume()` if the timeout argument is nonzero.

unspecified [24], and it is our experience that glibc's Linux implementation is unreliable in this case. As a workaround, we manually raise a signal on the thread and restore the context the same way we restored the snapshot when pausing, by overwriting the signal handler's context.

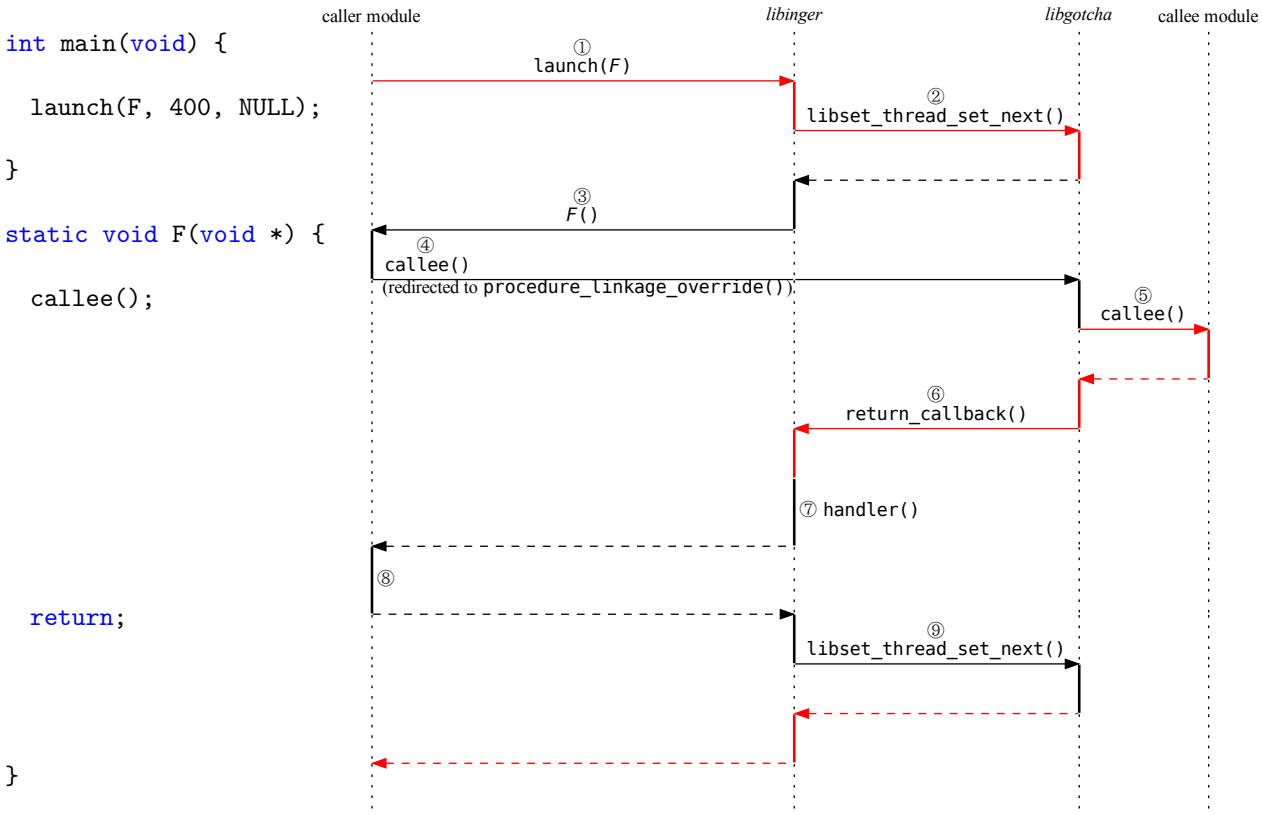
## 5.8 Memory isolation and deferred preemption

There is one final resource that *libinger* allocates for each preemptible function: a libset (Section 3.3). Here again, it uses a pool allocator so that each is reused automatically once no longer claimed by any preemptible function's `Continuation`. Recall that a libset represents a copy of the program's other dynamically-linked modules, and is used to isolate these modules' nonreentrant state by selectively relinking module entry points against this copy if they occur within the preemptible function (Section 3.5). Where this approach applies, it happens transparently to *libinger*.

Recall also that some functions cannot be duplicated in this manner, and instead require preemption to be deferred until the conclusion of their invocation. The *libinger* implementation observes the selective linking rule that the starting libset is uninterruptible (Section 3.6). Before pausing a preemptible function, the handler checks whether the next libset is equal to the starting libset. If so, it blocks the preemption signal and returns without affecting execution, thereby deferring preemption. When the running uninterruptible function returns, *libinger* receives a notification via a control library callback (Section 3.6.2) and unblocks the signal again. To guard against preemptible functions overrunning their deadlines by calling uninterruptible functions in a loop, this callback also immediately invokes the preemption handler, which pauses the function if it has timed out during the uninterruptible region.

To illustrate the division of responsibility between *libinger* and *libgotcha*, Figure 5.4 shows an example program that calls a preemptible function, alongside the resulting control transfers between modules.[6] ① The `main()` function calls `launch()` to invoke `F()` as a preemptible function. ② Among `launch()`'s responsibilities are switching out of the starting libset and enabling the preemption signal. At this point, the preemption handler starts executing periodically and checking whether the preemptible function is out of execution time. (If it ever detects a timeout, it will pause the preemptible function as described in Section 5.7, effectively skipping to step ⑨.) ③ Its setup work complete, `launch()` invokes the preemptible function. ④ This preemptible function happens to call another function `callee()` located in a different module, so *libgotcha* intercepts the call for selective relinking. There are two possibilities for what happens next: either `callee()` is an ordinary function (in which case *libgotcha* routes the call to the libset's local copy and we skip to step ⑧ once it returns), or it is an uninterruptible function. ⑤ In the latter case, before transferring control to `callee()`, *libgotcha* automatically switches to the starting libset, deferring preemption (Section 3.6). ⑥ When `callee()` returns, *libgotcha* invokes *libinger*'s callback, which switches back to the preemptible function's libset and reenables the preemption signal. ⑦ The callback raises the preemption signal, and the handler checks whether the preemptible function has timed out (skipping to step ⑨ if so). ⑧ The function call has completed without timing out, so *libinger* returns control to the preemptible function. The preemptible function happens to return, transferring control back to `launch()`. ⑨ Before returning to `main()`,

[6] We depict *libinger* and *libgotcha* as separate modules for clarity. However, recall that *libinger* is actually located in the same module as *libgotcha* because it is an internal control library (Section 3.7).




**(a)** Example program **(b)** Corresponding inter-module control transfers (`callee()` is uninterruptible)

**Figure 5.4:** Cross-module function calls under *libinger*. Solid lines represent function calls; dashed ones represent returns. Color indicates uninterruptible code (i.e., next libset = starting libset).

`launch()` switches back to the starting libset, disabling preemption.

### 5.8.1 Starting libset exit analysis

The preemptible function abstraction permits an optimization that reduces the runtime overhead of selective relinking. If we can prove that a library can never call `launch()`, we also know that it can never cause a switch out of the starting libset, so *libgotcha* does not need to intercept any function calls or global variable accesses made by the copy of it in the starting libset.

In contrast to the control libraries featured in Chapter 4, *libinger* implements a new abstraction rather than reusing a well-known API surface. Recall from Section 3.3 that a preemptible function, like any isolated task, always runs with its own private next libset. Furthermore, the fact that the only way to create a new preemptible function is by calling the `launch()` wrapper function means that there is also only one way for a preemptible function to run with its current libset set to the starting libset: if it is invoked from the same module that defines it (since otherwise the caller of the wrapper function would see the function's symbol resolve to a PLOT stub that would update its current libset from its next one upon invocation). Isolated tasks running with their current libset set to the starting one are the only code regions that can cause

an automatic switch *out of* the starting libset (if they call an interruptible function in any other module). We know that a preemptible function with this property can only exist in a module that contains a call to `launch()`, so we only have to update the real GOT entries of those starting libset modules that declare a dependency on `libinger.so`.

The performance implications of this insight are significant. By exempting most of the starting libset from the runtime overheads of selective relinking, it eliminates all of the global variable interceptions that would otherwise occur on the critical path of `launch()` and `resume()`. Furthermore, it eliminates most or all global variable interceptions that would occur during other uninterruptible function calls, reducing the risk of significantly exceeding a timeout. And as we saw in Section 3.9, each global variable interception incurs several microseconds of latency.

We call the optimization starting libset exit analysis, and it should be equally applicable to other novel task abstractions. A general rule of thumb is that it applies to any control library whose benefits would not be reaped merely by preloading it.

## 5.9 Calls and returns

When invoking the preemptible function, there is a possibility that it will throw an exception. Most of *libinger* is implemented in Rust, but the process of switching stacks creates control transfers through C code when calling and returning from a preemptible function. It is undefined behavior for Rust panics to cross these call boundaries [56], so *libinger* must handle exceptions specially. When calling into the preemptible function, `launch()` stops exception unwinding by wrapping the call in the standard library's `catch_unwind()` function. When a preemptible function runs to "completion," `launch()` and `resume()` check whether it threw an exception and call `resume_unwind()` if so. It is possible that applications seeking to limit functions' execution time might also want to prevent them from crashing the thread, so one possible revision to the interface would be to provide a configuration option to skip this rethrow.

The other state that must be transferred between stacks is the preemptible function's return value, so `launch()` and `resume()` store it in an optional type before switching stacks. As explained in Section 5.5, each stack switch returns to the same point regardless of whether the function ran to completion or timed out and became paused. To disambiguate these cases, `launch()` and `resume()` check whether they have stored a return value. If so, they return it; otherwise, they package a `Continuation` object containing the paused function's execution state.

## 5.10 Application compatibility

To assess the extent to which *libinger* and *libgotcha* break existing code, we ran the Gnulib test suite, which exercises hundreds of POSIX and ISO C library functions. We ran each test in the suite within a preemptible function by preloading a library that wrapped `__libc_start_main()` and `launch()`ed the program's real `main()` with an infinite timeout. On a glibc 2.29 system, we currently pass approximately 495 of the 519 supported tests (give or take one or two flaky tests that rely on precise timing behavior affected by our high-frequency timer signals).

Of the roughly 25 failing tests, most center around functions that we have not prioritized supporting within preemptible functions: 7 use `pthread_create()` and other thread spawn functions, 6 use `fork()` and other process creation functions, and 4 use functions from the `exec()`

family. We have not decided what behavior is desirable when a preemptible function invokes these functions, but the simplest option is to disallow their use by defining enforced interposition replacement functions (Section 3.7.1) that return a failure code when called from within a preemptible function.

Three other recurring issues we encountered were conflicting uses of signals, replacement of our signal handlers, and interrupted system calls not restarted by `SA_RESTART` (Section 5.6). The former included tests using historical interfaces such as `alarm()` that use the same `SIGALRM` signal that we allocate for preempting the first preemptible function. This is not unexpected because *libinger* does not currently ensure that its preemption signals are not also used by the program. The handler replacement issues mostly involved tests removing our preemption handler by restoring the default signal disposition. This would cause the program to crash when the preemption signal came in, so we worked around it by providing a replacement `signal()` wrapper that ignored requests for the `SIG_DFL` disposition. It is never desirable for a preemptible function to interfere with its own preemption signal, so a more robust solution would be to add a replacement wrappers that were aware of its assigned signal and reported failure on all requests to reconfigure it. The most frequently recurring functions that did not respect `SA_RESTART` were `sleep()`, `usleep()`, and `nanosleep()`, which when interrupted return a "success" status and the remaining duration they would have continued to sleep for. We added replacement wrapper functions to call them repeatedly using this information. One function that returns the `EINTR` interrupted error code even under `SA_RESTART` is `select()`, so we added a replacement wrapper that runs it in a loop as long as this happened. There are other functions with atypical interruption behavior, and our system would benefit from systematically adding additional wrappers to hide this behavior [62].

A final interesting complication we encountered is that glibc functions that automatically load supporting dynamic libraries at runtime are not functional by default in manually-loaded linker namespaces. For instance, calling the `iconv()` family of character set conversion functions causes libc to attempt to load multiple libraries, each of which performs pairwise conversions between two specific character sets. The library attempts to find a minimal sequence of available pairwise conversions that provides a route between the requested source and destination character sets; to test whether each candidate pairwise conversion is supported, it attempts to load a formulaically-named shared library that may or may not exist. This means that it is expected that `dlopen()` may fail, in which case the dynamic linker will notify `libc.so` by calling its `_dl_signal_exception()` error-handling function. This function's default action is to abort the process, so in the case of `iconv()`, libc updates some internal state to indicate that this behavior should be suppressed if the call fails. Unfortunately, the dynamic linker always invokes `_dl_signal_exception()` in the main namespace. In our case, this means that if a preemptible function calls `iconv()`, the call is routed to the copy of `libc.so` in its own libset, which updates its internal state and then calls into the dynamic linker, which in turn calls into the starting libset's `libc.so` that is still configured to crash on failure. We work around the problem by defining a replacement `_dl_signal_exception()` wrapper that checks the old value of next libset and reroutes the call to that libset's libc. The problem and this solution apply to several other glibc features besides `iconv()` (Section 3.6.1).

We are confident that with additional engineering effort, it would be possible to pass the full test suite with the possible exception of the few tests that rely on precise timing characteristics or incorrectly assume that short blocking file descriptor I/O operations will not return short counts.

## 5.11 Cancelling a paused preemptible function

Should a caller decide not to finish running a timed-out preemptible function, it must deallocate it. In Rust, deallocation happens implicitly via the `Linger` type's destructor, whereas users of the C interface are responsible for explicitly calling the `cancel()` function on the `linger_t` instance.

As discussed in Chapter 3, *libgotcha* returns a preemptible function's libset to the pool for reuse when that function returns normally. However, when a function is cancelled before it finishes, none of the modules in its libset is safe to reuse in general: a library function might have been in the middle of executing. To avoid future problems with the libset, as part of a cancellation, *libinger* instructs *libgotcha* to reinitialize the function's libset before returning it to the pool.

Cancellation cleans up *libinger* resources allocated by `launch()`; however, the current implementation does not automatically release resources already claimed by the preemptible function itself. Instead, the preemptible function author must write a cleanup handler (Section 4.2) and invoke it immediately before any call to `cancel()`. We have, however, created a prototype demonstrating the feasibility of automatic cleanup in RAII languages such as Rust, which we detail in Chapter 6.

## 5.12 Evaluation

Table 5.1 shows the overhead of *libinger*'s core functions. Each test uses hundreds of preemptible functions, each with its own stack and continuation, but sharing an implementation; the goal is to measure invocation time, so the function body immediately calls `pause()`. We show the latencies with and without the use of dedicated TCBs and TLS areas for each preemptible function (Section 5.3). For comparison, we also measured the cost of calling `fork()` then `exit()`, and of calling `pthread_create()` with an empty function, while the parent thread waits using `waitpid()` or `pthread_join()`, respectively.

The results show that, as long as preemptible functions are allowed to run to completion, invoking them incurs a fraction of the latency of spawning a kernel thread, and an order of magnitude reduction over forking a process. We collected these measurements on the same machine using the same software versions as in Section 3.9. Note that the overheads from that section also apply to any program using preemptible functions, although the optimization from Section 5.8.1 mitigates the runtime latency effect on many of the function calls and global variable accesses that occur outside any preemptible function.

### 5.12.1 Cancellation response time

Unlike state of the art approaches, lightweight preemptible functions support cancellation.

To demonstrate this feature, we consider decompression bombs, files that expand exponentially when decoded, consuming enormous computation time in addition to their large memory footprint. PNG files are vulnerable to such an attack, and although *libpng* now supports some mitigations [52], one cannot always expect (or trust) such functionality from third-party code.

We benchmarked the use of *libpng*'s "simple API" to decode an in-memory PNG file. We then compared against synchronous isolation using preemptible functions, as well as the naïve alternative mitigations proposed in Section 2.1. For preemptible functions, we wrapped all uses

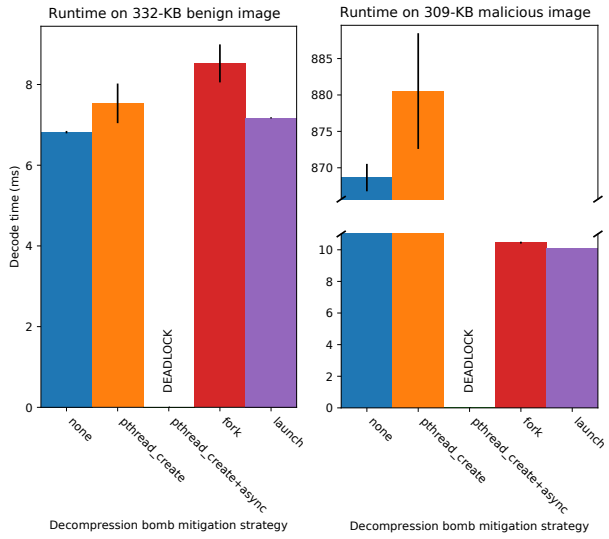
| Operation | Time with TLSes ($\mu s$) | Time without TLSes (µs) |
|---|---|---|
| `launch()` | 11.0 ± 0.5 | 8.0 ± 0.7 |
| `resume()` | 10.4 ± 0.4 | 6.9 ± 0.2 |
| `cancel()` | 42.0 ± 3.7 | 42.1 ± 1.8 |

**(a)** Preemptible function operations, with and without per-function TLS areas

| Operation | Time (µs) |
|---|---|
| `fork()` | 686 ± 21 |
| `pthread_create()` | 68 ± 18 |

**(b)** Process and thread spawns without *libinger*

**Table 5.1:** Latency of preemptible function interfaces




**(a)** Benign image **(b)** Malicious image

**Figure 5.5:** *libpng* in-memory image decode times

of *libpng* in a call to `launch()` and used a dedicated (but blocking) reaper thread to remove the cost of cancellation from the critical path; for threads, we used `pthread_create()` followed by `pthread_timedjoin_np()` and, conditionally, `pthread_cancel()` and `pthread_join()`; and for processes, we used `fork()` followed by `sigtimedwait()`, a conditional `kill()`, then a `waitpid()` to reap the child. We ran `pthread_cancel()` both with and without asynchronous cancelability enabled, but the former always deadlocked. The timeout was 10 ms in all cases.

Running on the benign RGB image `mirjam_meijer_mirjam_mei.png` from version `1:0.18+dfsg-15` of Debian's `openclipart-png` package showed `launch()` to be both faster and lower-variance than the other approaches, adding 355 $\mu$s or 5.2% over the baseline (Figure 5.5a). The results for `fork()` represent a best-case scenario for that technique, as we discarded the result buffer rather than implementing a shared memory mechanism to transfer its ownership to the rest of the application, and as the cost of the system call will increase with the number pages

mapped by the process (which was small in this case).

Next, we tried a similarly-sized RGB decompression bomb from revision `b726584` of `https://bomb.codes` (Figure 5.5b). Without asynchronous cancelability, the pthreads approach was unable to interrupt the thread. Here, `launch()` exceeded the deadline by just 100 $\mu$s, a figure that includes deviation due to the 100-$\mu$s preemption interval in addition to *libinger*'s own overhead. It again had the lowest variance.

Applying preemptible functions proved easy: the `launch()`/`cancel()` approach took just 20 lines of Rust, including the implementation of a reaper thread to move libset reinitialization off the critical path. In comparison, the `fork()`/`sigtimedwait()` approach required 25 lines of Rust. Note that both benchmarks include unsafe Rust (e.g., to use the *libpng* C library and zero-copy buffers).

We ran this experiment on an Intel Xeon E5-2683 v4 (Broadwell) server clocked at 2.1 GHz and running Linux 4.12.6, rustc 1.36.0, and glibc 2.29. As this kernel did not provide mitigations for the Meltdown or Spectre side-channel attacks, this configuration is especially favorable for `fork()` and `pthread_create()` latencies. We used an older version of *libinger* predating support for preemptible function–specific TLS and with *libgotcha*'s global variable interception disabled. This build relied on our early, higher-latency approach to libset reinitialization (Section 3.4), hence our use of a dedicated reaper thread.

> “ And with that Glóin embarked on a long account of the doings of the Dwarf-Kingdom. He was delighted to have found so polite a listener... ”
>
> — J. R. R. Tolkien, *The Fellowship of the Ring*

# Chapter 6

# Resource cleanup and async unwinding: the *ingerc* compiler

As described so far, one of the facilities that *libinger* enables is asynchronous function cancellation. As we saw in Chapters 2 and 4, this is a significant achievement that is only possible under the POSIX safety model thanks to selective relinking. However, one missing piece of functionality is automatic cleanup of any resources the cancelled function had allocated.

The resource leaks associated with cancelling a function are a significant problem: they make cancellation infeasible for long-running applications, which would experience the cumulative leakage of the resources allocated by all such cancelled functions. While a garbage collector would be able to find the leaked resources, deallocating them might still prove challenging because, without a record of the interruption point where cancellation occurred, it would not be safe to run object finalizers. Of course, our system targets unmanaged languages, so we must accomplish resource cleanup without a garbage collector.

## 6.1 Languages with unstructured resource management

In languages such as C, resource lifetimes are completely unstructured, with each allocation and deallocation performed via an ad-hoc function call. Some such functions are well-known because they are prescribed by the C and/or POSIX standards: `malloc()`/`free()`, `open()`/`close()`, etc. However, applications and libraries can provide their own resource-allocation interfaces, so it is not possible to identify or track resource management in general. Worse, there is no standardization of deallocation functions' interface. These language properties mean that automating cleanup would require hand-annotating all custom allocation and deallocation functions throughout the application and its dependencies; such annotations would have to provide associations between each allocator and its corresponding deallocator, as well as information about how to call the latter.

Were one to build a system to support this, one would need to use an approach like that of Valgrind's Memcheck [60] or LLVM's AddressSanitizer [59], which instrument the application's allocation and deallocation calls. Neither system could be imported wholesale: Both assume at a design level that memory is the only resource whose allocations are being tracked. Valgrind depends on expensive dynamic instruction translation that is not suitable for production use. AddressSanitizer does not track how each resource was allocated unless paired with the separate

MemorySanitizer system [64] run in origin-tracking mode; this adds another 2–7x slowdown on top of AddressSanitizer's own 2x execution time penalty.

For rolling one's own allocation and deallocation tracker, *libgotcha*'s existing ability to intercept function calls might prove useful. The bookkeeping structures would need to be mutable, so care would have to be taken to avoid designing around data structures with amortized time complexities, as this would introduce undesirable unpredictable pauses in preemptible function execution reminiscent of garbage collection.[1] For instance, storing allocation records in a hash table would require periodic rebalancing.

Because of the above limitations, we have not pursued automatic resource cleanup for preemptible functions written in C. Developers of long-running C applications should always write a cleanup handler for each preemptible function they might need to cancel (Section 5.11).

## 6.2 Languages following the RAII principle

The situation is more promising in Rust. Like C++, it adheres to the RAII (Resource Allocation Is Initialization) idiom that associates each resource's lifetime with that of some object. Whenever an object goes out of scope, the program invokes its destructor and those of its members, freeing the associated resources. Thus, the problem of releasing the resources associated with a cancellation can be reduced to that of invoking the destructors of the objects that are alive at the interruption point. Notice that, in contrast to garbage collection, such a model does not divorce the problem of deallocation from the cancelled function's code; as such, it is not subject to the safety problems of invoking finalizers, as only the destructors of objects whose initialization is already complete can be invoked.

Faced with the challenge of safely preempting in the presence of shared state caused by non-reentrant library interfaces, we found that we could leverage dynamic linking to solve the problem automatically, and built the *libgotcha* runtime to do just that. Here again, we are fortunate to find an existing runtime facility that can be repurposed to call destructors at an arbitrary position in the program: the Rust language already supports exceptions (which it calls "panics"). One significant advantage to building on top of exceptions rather than implementing separate resource tracking is that exception handling is already designed to add no overhead to the non-exceptional execution path. With the exception of adding one function call to each function that owns objects with destructors, we believe it is possible to provide automatic cleanup without imposing runtime overhead on tasks that are never cancelled.

## 6.3 A brief tour of exception handling

Whenever a program throws an exception, the language runtime must find the point in the program that handles that exception. To prevent resource leaks, deadlocks, and other bugs, it must then invoke the destructors of all objects that are in scope at the point where the exception was thrown, but out of scope at the point where it is caught. This feature of exception handling is perfectly suited to our use case.

[1] The *libgotcha* runtime itself does not suffer from this problem because its symbol lookup tables are immutable once process initialization is complete.

It is possible for a function to throw an exception that is then caught by one of its callers, so the language runtime must be able to "unwind" the stack, locating the stack frame of each function's caller. Code for the x86 architecture used to maintain a frame pointer that made it easy to find the bounds of a function's stack frame, but with the advent of x86-64, this is no longer standard; thus, the runtime needs some other way to find the next frame. Debuggers have long faced this very problem on other architectures, and the common approach is to rely on extra debugging information stored in the executable or library on disk. On Unix operating systems, most debuggers use the CFI (Call Frame Information) facility of the standard DWARF debugging format [19].

Modern exception runtimes repurpose this debugging information to unwind the stack once an exception has been thrown. The compiler produces the requisite information by generating CFI pseudoinstructions, which the assembler then transcribes into DWARF format and stores in the `.eh_frame` section of the object file. This section is present in non-debug builds and stripped object files and gets loaded into the process's memory image by the ELF loader or dynamic linker, in contrast to the CFI's more traditional home, the `.debug_frame` section. With the complexity of this approach comes the advantage that the application no longer has to update frame pointers during normal execution.

Call Frame Information alone is not a sufficient primitive to implement exception handling: the runtime must also be able to find the exception handler(s) present in each call frame and the destructors to invoke based on where in the function the exception was thrown. The compiler must supply this information, which it does by emitting pseudoinstructions that describe a metadata region known as the LSDA (Language-Specific Data Area); the assembler stores this in the object file's `.gcc_except_table` section. For each function, the LSDA contains a table mapping instruction address ranges to landing pads, code regions within the function that serve either to catch exceptions or to invoke destructors. Our discussion will focus on the latter type, known as cleanup landing pads.

## 6.4 Asynchronous exception handling

Because exceptions are generated synchronously, they can only occur on calls to functions that can throw. Since compilers know which functions can throw, they generally only output LSDA entries that are accurate for those functions' call sites. But since *libinger* interrupts functions preemptively, we need to trigger unwinding and cleanup at whatever arbitrary point the function was paused at before being cancelled.

Triggering unwinding is a simple matter of tweaking the stack pointer and instruction pointer of the preemptible function to be cancelled in order to forge a call to a function that raises an exception using Rust's `panic!()` macro. But providing instruction-accurate cleanup information requires us to address the following challenges:

1. **Optimized builds remove some functions' LSDA tables and landing pads.** We have noticed that enabling optimizations via the Rust compiler's `-O` switch causes some functions that have exception-handling support in debug builds to instead be compiled without it. We describe our workaround for this issue in Section 6.4.1.

2. **Functions that "return" values via pointer parameters lack exception-handling in-**

**formation.** We have noticed that such "sret" functions tend to lack any exception information at the LLVM IR level, even if they operate on objects with destructors. This is a problem because, although the objects exist in the caller's stack frame, they must still be treated as owned by the function that is "returning" them, so that we will clean them up if cancellation occurs between the time they are allocated and that function returns. Such functions are more common than one might expect and include most constructors: the Rust compiler prefers to compile functions that return large objects in this manner to avoid moving them to the caller's stack frame immediately afterward. See Section 6.4.2.

3. **Many LSDA entries associate the landing pad with too few instructions following or preceding a function call site.** Injecting an exception in such execution regions results in leaks or deallocating before allocation, respectively. Our investigation revealed that these discrepancies result from changing instruction boundaries during lowering from LLVM IR to the platform's assembly language; in particular, the backend does not account for the `mov` and `lea` instructions that perform argument passing before most `call`s. See Section 6.4.3.

4. **The runtime does not discriminate between being in the middle of executing a function and having just retired its `ret` instruction and jumped back to the call site.** In either case, it will not invoke any cleanup landing pads in the caller. The two scenarios are indistinguishable under the assumption that no exception can occur at these points in the function. However, the fact that we can inject one there creates an important distinction for our purposes: until the function returns, it still has ownership of its live variables and its landing pads are responsible for cleaning them up, whereas after it has returned, it is impossible for those landing pads to be invoked and cleanup must necessarily be up to the caller. See Section 6.4.4.

5. **Unwinding on the first instruction of a function fails because the runtime consults the LSDA table for the function whose definition precedes it in memory.** This issue turns out to have the same cause as the previous one, but the two situations demand different solutions. See Section 6.4.5.

6. **Performing function calls during the prologue or epilogue of a function is unsafe.** The x86-64 ABI (Application Binary Interface) specifies that the stack must always be 16-byte aligned before calling a function, which is not true until the function has reserved space for an odd number of 8-byte values (excluding the return address) in its stack frame. See Section 6.4.6.

7. **If attempted on the instruction just after one that repositions the stack pointer, unwinding miscalculates the frame address.** While this behavior appears consistent between the libgcc and libunwind (LLVM) unwind implementations, we suspect it exists because exceptions ordinarily never occur in the prologue or epilogue of the function. GCC has an `-fasynchronous-unwind-tables` switch that is intended to make the frame information accurate down to the instruction, but Clang only includes this switch for command-line compatibility and doesn't actually implement this feature. As a likely consequence of this lack of support from the LLVM project itself, the Rust compiler also makes no attempt to offer it. See Section 6.4.7.

8. **Cleanup landing pads do not work reliably if associated with the function epilogue.** This happens because the epilogue adjusts the stack pointer, in many cases causing any synthetic function call (e.g., to inject an exception) to clobber the very stack values the landing pad is trying to clean up. Incidentally, a Web search for "LLVM unwind function epilogue" reveals that the unwind info is not trustworthy during the epilogue in the general case. Indeed, there have been several patchsets attempting to fix this, some of which were merged, but each of which was subsequently reverted for breaking some other architecture. So it would appear not only that this is the primary design issue blocking LLVM support for asynchronous unwind tables, but also that we must avoid injecting exceptions in epilogues altogether. See Section 6.4.8.

Rather than integrate a fully general resource cleanup solution into *libinger*, we have prototyped the components to solve these problems and used these to build a proof of concept implementation of the compiler transformations necessary to support asynchronous exception handling. This prototype represents preliminary evidence that our approach is feasible, although it would take additional engineering effort to achieve compatibility with nontrivial applications.

The below numbered sections describe our approach to solving each of the challenges listed above. The product of the work described in this section is a shell script, *ingerc*, that wraps rustc and applies all the described transformations to produce an output program or library ready for runtime-assisted cancellation cleanup.

### 6.4.1 Skipping optimization passes that remove exception handling

Testing with rustc 1.56.0, we have found that the `prune-eh`, `function-attrs`, and `inline` LLVM optimization passes are responsible for stripping the LSDA tables and landing pads from some functions in optimized builds. We have developed a shell script to invoke rustc without these passes, a task that is unfortunately complicated by the compiler's command-line interface, which only accepts a list of all the passes to run.

We recognize that disabling the inlining pass is likely to reduce the efficiency of compiled code, but we leave it to future work to investigate why this pass is removing exception information from functions otherwise unaffected by inlining.

### 6.4.2 Adding exception-handling support to functions' LLVM IR

The above script does not address functions for which the compiler emits no exception-handling information even in debug builds. As before, this problem is easiest to address in the intermediate representation, where the addition of an exception-handling personality and a `landingpad` instruction will cause the LLVM backend to emit an LSDA table and landing pad for the function.

To reduce implementation complexity, we do not attempt to detect which functions own objects with destructors, and instead introduce exception handling into any functions that do not already have it. This saves us from having to query complex properties of the IR and reduces our task to one of simple text transformations. We implement these in a TypeScript script performing regular expression replacements.

The landing pads we insert at this stage are empty skeletons that do not actually invoke any destructors. We describe how we identify which destructor(s) to invoke (if any) and add the calls at the end of Section 6.4.3.

### 6.4.3 Adjusting LSDA entries

The possibility that cancellation injects an exception during the argument-passing instructions preceding a call violates a design assumption of LLVM's LSDA generation. IR instructions such as `call` often expand to multiple machine instructions, most commonly to perform argument passing before the function dispatch. However, the backend generates the address ranges for LSDA entries using labels in the IR. This means that ordinary optimization and transform passes cannot associate landing pads with some but not all of the instructions comprising a function call sequence.

To get around this problem, we had to implement a plugin that loads a code generation pass into `llc`, the LLVM static compiler. The pass works at the x86-64 machine instruction level to reposition LSDA-related labels and resize the code region on the normal execution path with which a cleanup landing pad is associated. To prevent leaks, if the ending label falls before a destructor call, the pass moves it downward to just before the machine `call` instruction; otherwise, the pass moves it downward to just before the function epilogue. To prevent issuing destructor calls before construction, if the starting label falls before the function call that produces the object to be cleaned up, the pass moves it downward to just after that call.

The pass also identifies functions with parameters annotated as `sret` in the LLVM IR. These correspond to functions where the script from Section 6.4.2 added synthetic landing pads. The pass checks to see whether the involved type(s) have destructors; if so, it adds destructor calls to the landing pad.

### 6.4.4 Detecting whether a function has just returned

Regardless of whether a function has returned, LLVM's libunwind treats the caller frame as sitting within the call instruction, rather than on the subsequent instruction located at the return address. Here is the offending libunwind code:

```
// If the last line of a function is a "throw" the compiler sometimes
// emits no instructions after the call to __cxa_throw. This means
// the return address is actually the start of the next function.
// To disambiguate this, back up the pc when we know it is a return
// address.
if (isReturnAddress)
     --pc;
```

Since we propose to inject the exception by forging a function call, libunwind always assumes the frame where we did this is a return address and performs the decrement. The obvious workaround would be to remove this code from libunwind, at the cost of potentially breaking unwinding through C++ code that might be present in the program. Indeed, a glance through the disassembly of the Rust standard library shows that rustc emits `ud2` (invalid) instructions following the call sites in the scenario described in the comment, so Rust code is unaffected by the problem.

However, it turns out that the above code has another important effect beyond that documented in the comment: it avoids running the cleanup landing pad associated with the code region following the call site if the called function was still executing at the time the exception

was thrown. This is essential because in this case, the called function still has ownership over any objects requiring cleanup, and their state is undefined from the perspective of the caller. The safe and correct thing for the runtime to do is to invoke the called function's landing pad but not the caller's.

For this reason, we need to override this libunwind behavior only at the instruction where we injected the exception, and only if that instruction immediately follows a call (so that libunwind would confuse the situation with one where the called function was still executing). We propose to accomplish that by applying a heuristic-based tweak just before the *libinger* cancellation code injects the exception call: if the instruction pointer is equal to the 8-byte value offset −8 bytes from the stack pointer, the function return just completed and we should add one to the instruction pointer to counteract the described libunwind behavior for this stack frame only. We have prototyped this technique in a GDB script, allowing us to test it at any arbitrary instruction within a simulated preemptible function.

### 6.4.5 Unwinding from the first instruction of a function

Another consequence of the libunwind implementation detail described in Section 6.4.4 is that unwinding with the instruction pointer positioned on the first instruction of a function results in an address associated with the preceding function in memory. Therefore, the runtime does not find the correct LSDA for the function (if it finds one at all).

It is hard to detect this problem without consulting the LSDA, so we implement a fix by patching libunwind, which already decodes this information. We insert a check whether the current instruction pointer falls at the very beginning of its function; if so, we set the `isReturnAddress` flag to skip the instruction pointer adjustment.

### 6.4.6 Calling functions when the stack is misaligned

Our standard cancellation response of forging a call to a function that panics causes crashes when the stack is misaligned, as during a function prologue. Fortunately, we can easily solve this by instead selectively calling a function that does not allocate any space in its own stack frame. This has the effect of restoring the stack alignment (because of the return address pushed by the `call` instruction) before calling any complex code that relies on alignment. Crucially, it does so without introducing any invalid stack frames that would break unwinding. The following function suffices:

```
        .globl realign
realign:
        .cfi_startproc
        call panic
        ud2
        .cfi_endproc
```

(where `panic` is the function that would ordinarily inject the exception). We have tested this solution in GDB's scripting language.

### 6.4.7 Unwinding after an instruction that moves the stack pointer

To compute the offset from the stack pointer to the return address, libunwind contains a function called `parseFDEInstructions()`. It loops through the CFI instructions in the `.eh_frame` section, continuing as long as `codeOffset < pcoffset` to process the stack pointer adjustments for the instructions that have executed so far. Unfortunately, this appears to fall one CFI instruction short when the instruction that has just retired repositioned the stack pointer. Changing the < to a <= fixes the problem.

Section 6.4.3 of the DWARF specification [18] seems to agree with this sign change. We hypothesize that libunwind inherited this off-by-one error from libgcc in its effort to replicate the older library's behavior. The libunwind test suite continues to pass after making the change, suggesting that an incomprehensive test suite has allowed the mistake to avoid detection. Furthermore, Clang's lack of support for asynchronous unwind tables has probably prevented the community from encountering the unwind failures we have.

### 6.4.8 Unwinding in the epilogue of a function

As discussed in Section 6.4, function epilogues are perilous for exception injection, and even unwinding in them is currently unreliable. To work around these limitations, we have developed our own compiler-assisted runtime component.

That epilogues pop values off the stack might suggest that it is no longer possible to clean up a function's resources once its epilogue has started executing; fortunately, they have an important property that refutes this intuition. Although the epilogue removes elements such as saved register values from the stack, it only moves the stack pointer and does not overwrite the contents of the stack frame. Thus, if we could undo the epilogue's effects, we could then inject an exception and it would be handled as if the epilogue had never executed at all. This is precisely our approach.

To support time traveling backward to just before a function's epilogue, we need the program to record its instruction pointer just before it enters the epilogue. We accomplish this by having the script introduced in Section 6.4.2 and our LLVM pass cooperate to insert a call to a custom function, `ingerc_epilogue_start()`, before each function's epilogue(s).

In addition to saving the instruction pointer, `ingerc_epilogue_start()` informs *libinger* that an epilogue is currently running, so that cancelling a preemptible function will not inject an exception in the usual way. This means that we must also be able to inform *libinger* once the epilogue is finished, so `ingerc_epilogue_start()` overwrites the function's return address with the location of another function, `ingerc_epilogue_end()`, that performs this notification before returning to the real return address.

There are a few other values we need to save before starting the epilogue: (1) When `ingerc_epilogue_end()` runs, it will need to know the function's original return address. (2) If either of the functions we introduce at the beginning and end of the epilogue is running when the function is cancelled, the stack pointer will be different than at the point we intend to travel to, so we always store the original stack pointer as well. (3) To restore the return address in `ingerc_epilogue_end()` or when time traveling, we need the frame pointer. The frame address is not normally accessible at runtime, so we have our LLVM plugin insert code to pass it to `ingerc_epilogue_begin()`.

The functions we introduce run just before the function returns, so they must not overwrite any return registers. Because of this, we have hand coded them in assembly. We give their implementations in Listing 6.1. The `ingerc_epilogue_begin()` function saves all values into globals so they are accessible by the runtime. This allows us to use the stored return address (`ingerc_epilogue_ra`) to notify the runtime that the epilogue is currently executing, so we are careful to set that last and reset it to null in `ingerc_epilogue_end()`.

There is one other thing that these functions have to be careful about: if cancellation occurs while they are running but outside the region where `ingerc_epilogue_ra` is set, unwinding must be safe and invoke the correct cleanup landing pads. This is why `ingerc_epilogue_end()` returns to the real caller using a `push` and a `ret` instead of an unconditional branch. The `ingerc_epilogue_start()` implementation is compatible with ordinary unwinding, but its invocation can be troublesome because it is intentionally the last instruction before the epilogue. This would mean that cleanup in the caller frame would invoke the epilogue's landing pad, but it never has one. To prevent a leak in this situation, our LLVM plugin inserts a `nop` instruction after the call and before the epilogue label, in order to associate the function's return address with the landing pad for the preceding basic block.

## 6.5 Preemptible function cancellation

While we have not integrated resource cleanup support into *libinger*, our work on asynchronous exception handling suggests a design. We have prototyped the approach in isolation using a set of scripts that use GDB to interrupt execution after an arbitrary number of instructions have retired and inject an exception at that point. In this section, we give the algorithm and how it would integrate with the existing *libinger* codebase. We conclude by reasoning about its correctness based on where the preemptible function is in its execution at the time it is cancelled and discussing performance considerations.

Section 6.4 introduced our fundamental approach to asynchronous cleanup: the runtime should inject a synthetic exception at an arbitrary point in the preemptible function. It also presented a compiler wrapper script, *ingerc*, that applies a series of transformations to the code to make this safe and correct. In this section, we assume that the preemptible functions being cancelled are written in Rust, and that the application and all its Rust dependencies have been compiled with *ingerc* instead of rustc. We believe the latter requirement is reasonable because the Cargo build system already expects to have the source of all dependencies available. Indeed, new languages such as Rust and Go follow a growing trend of having unstable ABIs that preclude linking against precompiled build artifacts generated by a different compiler version.

Whereas the *libinger* C bindings implement the `cancel()` operation as a standalone function, the Rust interface performs cancellation in the destructor. Whenever a paused preemptible function goes out of scope, the destructor notices that it has not run to completion and reinitializes its libset to prepare it for reuse. Listing 6.2 gives pseudocode for a function that the destructor would call right before this reinitialization to clean up the preemptible function's own resources. This works because, for safety, `resume()` catches all panics before they can cross the FFI (Foreign Function Interface) boundary (Section 5.9). To prevent the Rust runtime from outputting a diagnostic message when the panic occurs, it is advisable to first disable the Rust standard library's panic handler in the preemptible function's libset. The standard library exposes the

```
        .globl ingerc_epilogue_start
ingerc_epilogue_start:
        # Save our return address as the destination instruction pointer.
        mov    ingerc_epilogue_ip@gotpcrel(%rip), %rsi
        mov    (%rsp), %rcx
        mov    %rcx, (%rsi)
        # Save the stack pointer as it was before we were called.
        mov    ingerc_epilogue_sp@gotpcrel(%rip), %rsi
        lea    8(%rsp), %rcx
        mov    %rcx, (%rsi)
        # Save the frame pointer, which we received as an argument.
        mov    ingerc_epilogue_fp@gotpcrel(%rip), %rsi
        mov    %rdi, %rcx
        mov    %rcx, (%rsi)
        # Save the return address of our caller.
        mov    ingerc_epilogue_ra@gotpcrel(%rip), %rsi
        mov    (%rdi), %rcx
        mov    %rcx, (%rsi)
        # Make our caller return to ingerc_epilogue_end().
        mov    ingerc_epilogue_end@gotpcrel(%rip), %rsi
        mov    %rsi, (%rdi)
        # Return.
        ret

        .globl ingerc_epilogue_end
ingerc_epilogue_end:
        # Put the original return address on the stack.
        mov    ingerc_epilogue_ra@gotpcrel(%rip), %rsi
        mov    (%rsi), %rdi
        push   %rdi
        # Clear the saved return address.
        xor    %rcx, %rcx
        mov    %rcx, (%rsi)
        # Return to the original caller.
        ret
```

**Listing 6.1:** Code to support time travel out of the epilogue. The `@gotpcrel` relocations are position-independent GOT lookups of the globals' addresses.

```
function cleanup(linger_t func):
        ucontext_t snapshot = func.continuation;

        // Check whether some callee just returned (section 6.4.4)
        uint64_t retaddr = 8 bytes preceding snapshot.uc_mcontext[REG_RSP]
        if snapshot.uc_mcontext[REG_RIP] == retaddr:
                increment snapshot.uc_mcontext[REG_RIP]

        // If in epilogue, time travel to before (section 6.4.8)
        if ingerc_epilogue_ra != NULL:
                snapshot.uc_mcontext[REG_RIP] = ingerc_epilogue_ip
                snapshot.uc_mcontext[REG_RSP] = ingerc_epilogue_sp
                location ingerc_epilogue_fp = ingerc_epilogue_ra
                ingerc_epilogue_ra = NULL

        if 16 divides snapshot.uc_mcontext[REG_RSP]:
                // Inject an exception using panic!() (section 6.4)
                snapshot.uc_mcontext[REG_RIP] = panic
        else:
                // Realign the stack and inject exception (section 6.4.6)
                snapshot.uc_mcontext[REG_RIP] = realign

        // Throw the exception and let the cleanup landing pads run
        resume(func, UNLIMITED_TIME)
```

**Listing 6.2:** Resource cleanup for cancelled preemptible functions (pseudocode)

`panic::set_hook()` function for doing this.

Table 6.1 summarizes our method of resource cleanup, showing the actions taken by our proposed runtime at each possible point the preemptible function (or any of its callees) might be paused when cancellation occurs. It can be seen that we have handled all possible points within the body of an ordinary function. The case we have scoped out of our investigation is cancelling a preemptible function while it is running a destructor; instead of attempting this, we suggest implementing a mechanism to detect this case (e.g., unwinding the stack or hooking into the Rust standard library) and using a return address trick similar to that from Section 6.4.8 to delay resource cleanup until the destructor has finished.

## 6.6 Performance considerations

While our approach mostly avoids adding operations to the common execution path, the exception is epilogue handling. To support that case, we add one function call and six global variable accesses. We saw in Chapter 3 that in a normal application, each of these operations has a negligible cost of just a few cycles. However, we also found that *libgotcha* slows down accesses to

| Position within running function | Indicator | Possible to unwind here? | Handling | |
|---|---|---|---|---|
| | | | Normal execution | When cancelling |
| First instruction | Stack alignment | No (stack misalignment) | – | § 6.4.5, 6.4.6 |
| Function prologue | Stack alignment | No (stack misalignment) | – | § 6.4.6 |
| Function body | – | Yes | – | Raise exception |
| After call site | Return address | Yes (but leaks) | – | § 6.4.4 |
| Before epilogue | – | Yes | § 6.4.8 | Raise exception |
| Function epilogue | Saved return address | No (calls could clobber stack frame) | – | § 6.4.8 |
| After return | – | – | § 6.4.8 | Raise exception |

**Table 6.1:** Cancelled function resource cleanup by position within the running function

dynamically-linked global variables considerably. Fortunately, the names of the symbols we introduce are well known, so when integrating the new runtime components, we could introduce special cases to prevent *libgotcha* from intercepting their uses. This would make the symbols local to each libset and thereby obviate the need to pay the expensive reference costs.

> “Who controls the past controls the future.
> Who controls the present controls the past.”
> — George Orwell, *1984*

# Chapter 7

# Preemptive userland threading: the *libturquoise* futures executor

Until now, we have limited our discussion to synchronous, single-threaded programs. In this chapter, we will show that the preemptible function abstraction is equally relevant to asynchronous and parallel programs, and that it composes naturally with both futures and threads. As a proof of concept, we have created *libturquoise*,[1] a preemptive userland thread library.

That *libturquoise* provides preemptive scheduling is a significant achievement: *Shinjuku* observes that “there have been several efforts to implement efficient, user-space thread libraries. They all focus on cooperative scheduling” [35]. (Though *RT* from Section 2.2 could be a counterexample, its lack of nonreentrancy support renders it far from general purpose.) We attribute the dearth of preemptive userland thread libraries to a lack of natural abstractions to support them.

Before presenting the *libturquoise* design, we begin with some context about futures.

## 7.1 Futures and asynchronous I/O

As mentioned in Section 2.1, futures are a primitive for expressing asynchronous program tasks in a format amenable to cooperative scheduling. Structuring a program around futures makes it easy to achieve low latency by enabling the runtime to reschedule slow operations off the critical path. Alas, blocking system calls (which cannot be rescheduled by userland) defeat this approach.

The community has done extensive prior work to support asynchronous I/O via result callbacks [42, 41, 43, 48]. Futures runtimes such as Rust’s Hyper [31] have adapted this approach by providing I/O libraries whose functions return futures. Rather than duplicate this work, we have integrated preemptible functions with futures so they can leverage it.

## 7.2 Preemptible futures

For seamless interoperation between preemptible functions and the futures ecosystem, we built a preemptible future adapter that wraps the *libinger* API. Like a normal future, a preemptible future yields when its result is not ready, but it can also time out.

[1] so called because it implements “green threading with a twist”

```
function PreemptibleFuture(Future fut, Num timeout):
        function adapt():
                // Poll wrapped future in the usual way
                while poll(fut) == NotReady:
                        pause()
        fut.linger = launch(adapt, CREATE_ONLY)
        fut.timeout = timeout
        return fut

// Custom polling logic for preemptible futures
function poll(PreemptibleFuture fut):
        resume(fut.linger, fut.timeout);
        if has_finished(fut.linger):
                return Ready
        else
                if called_pause(fut.linger):
                        notify_unblocked(fut.subscribers)
                return NotReady
```

**Listing 7.1:** Futures adapter type (pseudocode)

Each language has its own futures interface, so preemptible futures are not language agnostic like the preemptible functions API. Fortunately, they are easy to implement by using `pause()` to propagate cooperative yields across the preemptive function boundary. We give the type construction and polling algorithm in Listing 7.1; our Rust implementation is only 70 lines.

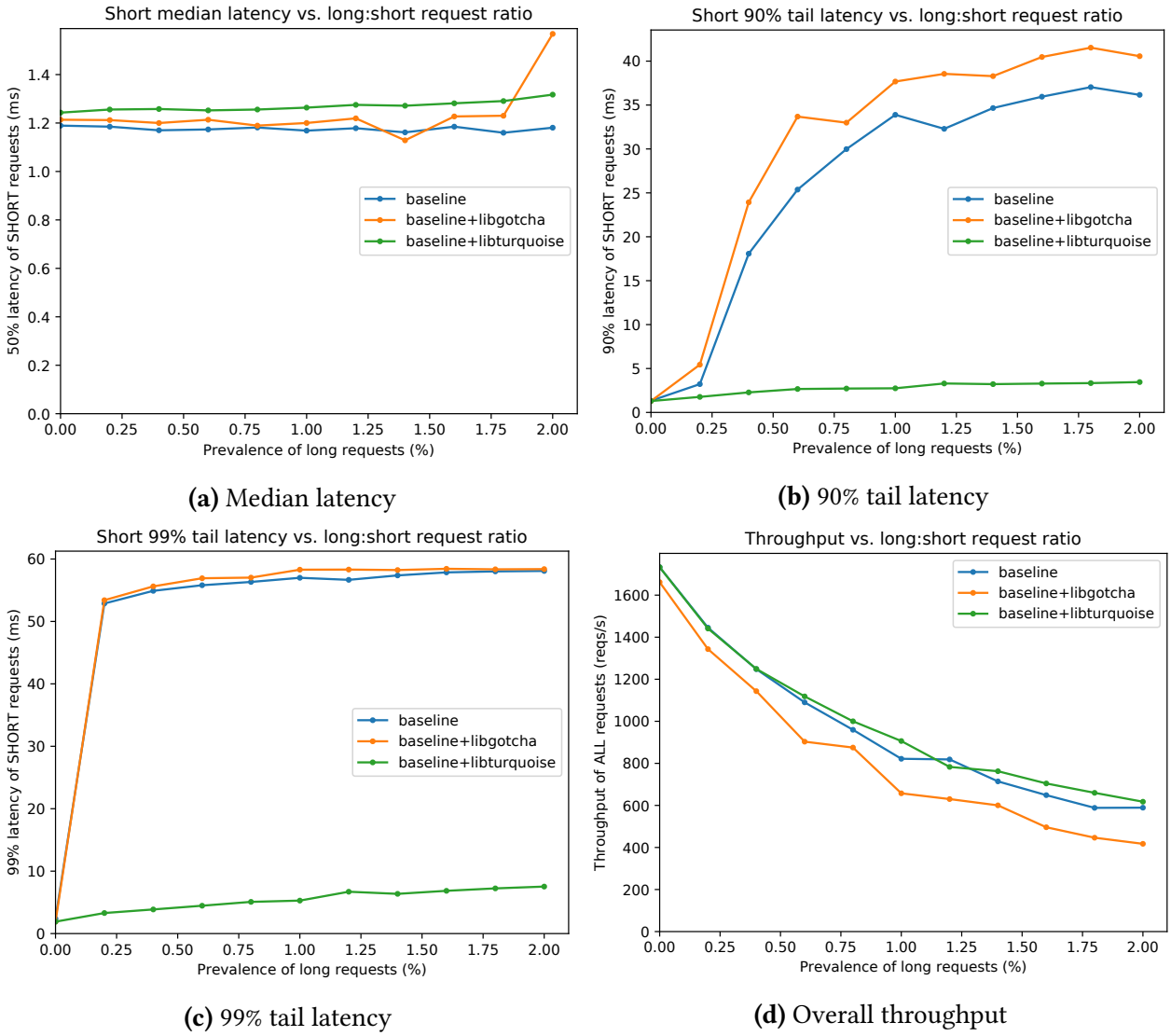
## 7.3 Preemptive userland threading

We built the *libturquoise* thread library by modifying the *tokio-threadpool* [68] work-stealing scheduler from the Rust futures ecosystem. Starting from version 0.1.16 of the upstream project, we added 50 lines of code that wrap each incoming task in a preemptible future.

Currently, *libturquoise* assigns each future it launches or resumes the same fixed time budget, although this design could be extended to support multiple job priorities. When a task times out, the scheduler pops it from its worker thread's job queue and pushes it to the incoming queue, offering it to any available worker for rescheduling after all other waiting jobs have had a turn.

## 7.4 Evaluation

To test whether our thread library could combat head-of-line blocking in a large system, we benchmarked *hyper*, the highest-performing Web server in TechEmpower's plaintext benchmark as of July 2019 [31]. The server uses *tokio-threadpool* for scheduling; because the changes described in Chapter 7 are transparent, making *hyper* preemptive was as easy as building against




**(a)** Median latency

**(b)** 90% tail latency

**(c)** 99% tail latency

**(d)** Overall throughput

**Figure 7.1:** *hyper* Web server with 500-$\mu$s (short) and 50-ms (long) requests

*libturquoise* instead. In fact, we did not even check out the *hyper* codebase. We configured *libturquoise* with a task timeout of 2 ms, give or take a 100-$\mu$s *libinger* preemption interval, and configured it to serve responses only after spinning in a busy loop for a number of iterations specified in each request. For our client, we modified version 4.1.0 of the *wrk* [72] closed-loop HTTP load generator to separately record the latency distributions of two different request classes.

Our testbed consisted of two machines connected by a direct 10-GbE link. We pinned *hyper* to the 16 physical cores on the NIC's NUMA node of our Broadwell server. Our client machine, a Intel Xeon E5-2697 v3 (Haswell) running Linux 4.10.0, ran a separate *wrk* process pinned to each of the 14 logical cores on the NIC's NUMA node. Each client core maintained two concurrent pipelined HTTP connections.

We used loop lengths of approximately 500 $\mu$s and 50 ms for short and long requests, respectively, viewing the latter requests as possible DoS attacks on the system. We varied the percentage of long requests from 0% to 2% and measured the round-trip median and tail latencies

of short requests and the throughput of all requests. Figure 7.1 plots the results for three server configurations: `baseline` is cooperative scheduling via *tokio-threadpool*, `baseline+libgotcha` is the same but with *libgotcha* loaded to assess the impact of slower dynamic function calls, and `baseline+libturquoise` is preemptive scheduling via *libturquoise*. A 2% long request mix was enough to reduce the throughput of the *libgotcha* server enough to impact the median short request latency. The experiment shows that preemptible functions keep the tail latency of short requests scaling linearly at the cost of a modest 4.5% median latency overhead when not under attack.

All experiments were run on an Intel Xeon E5-2683 v4 (Broadwell) server running Linux 4.12.6, rustc 1.36.0, gcc 9.2.1, and glibc 2.29. We used an older version of *libinger* without support for per-task thread-local storage (Section 3.7.2). This version exhibited the lower `launch()` and `resume()` latencies reported in our conference paper [8], as opposed to the latencies of the more feature-complete version benchmarked in Section 5.12; however, we expect the latest version to exhibit the same behavior, albeit with the knee of the latency curve at a different x value.

## 7.5 Conclusion

In this chapter, we used preemptible functions to implement both preemptible futures and a first-in-class preemptive user thread library with no dependency on custom kernel support. Making the thread library preemptive was transparent, with the resulting system exposing the same API surface as the (cooperative) upstream project. We demonstrated how such preemptive threading can mitigate denial-of-service attacks based on compute-bound requests.

While both artifacts created for this chapter are useful in their own right, they also serve to demonstrate that the preemptible functions abstraction composes with both futures and threads.

> “ ‘Do you really think I am so shortsighted? The Guild of Engineers plans further ahead than you suspect. London will never stop moving. Movement is life.’ ”
>
> — Philip Reeve, *Mortal Engines*

# Chapter 8

# Preemptible remote procedure calls: the *strobelight* caching RPC server

Whereas local function calls with timeouts have yet to become a mainstream idea, it has long been standard to use timeouts when making RPCs (Remote Procedure Calls) where a program on one machine invokes a function that executes on another. Because the connection to the remote machine could fail, it is common for the caller to specify a timeout after which the call should return an error code, allowing the client program to continue running. What these systems usually do not do is use this timeout to reduce wasted work on the server side. In their most classic formulation, RPCs do not even inform the server of the timeout; if the client times out, the server continues to work on the request, only to find once it has finished the work that the client is no longer listening for an answer. This is particularly wasteful if an ill-behaved client repeats a failing request with the same timeout, in which case the server duplicates the same work with the same likely fate. Newer systems might share the timeout with the server, but generally use it only for cooperative scheduling. Therefore, it is up to the developer of each server-side function to implement cancellation to prevent it from wasting resources serving doomed requests.

We had a first-year undergraduate student prototype a novel caching RPC server that addresses this limitation transparently to the server-side programmer. Over the course of two months, he built the *strobelight* RPC server, so called because it processes each request only while the client is listening for a response and pauses in the intervening intervals to avoid wasting server compute time. The system serves as an example of the new capabilities possible by using preemptible functions, and also a demonstration that preemptible functions are not only usable by expert programmers.

## 8.1 State-of-the-art RPC systems

The motivation for this work is that contemporary RPC systems do not generally support preemption. We begin with a brief survey of well-known systems and their approach to timeouts.

**ONC RPC** In the 1980s, Sun Microsystems developed a custom RPC protocol as part of the NFS (Network File System) project. The protocol has come to be known as ONC (Open Network Computing) RPC, and was standardized in a series of RFCs, most recently updated in 2009. The protocol is widely implemented, shipping with many Unix systems and Microsoft Windows. It is

```
if(context->IsCancelled()) {
        return Status(StatusCode::CANCELLED, "Deadline␣exceeded");
}
```

**Listing 8.1:** Checking the cancellation flag in a gRPC server-side function

minimal and leaves features such as timeouts to the transport protocol. The RFC notes that clients and servers must carefully handle the case of retrying timed out or otherwise failed requests to avoid executing the task multiple times [53].

**eRPC** The eRPC project achieves lower RPC latencies by providing low-level primitives tailored to userland networking drivers. Instead of Sun-style timeouts, it uses heartbeats to detect broken connections. Unrequited heartbeats cause the session to expire, in which case the server immediately notifies the client of the failure, but continues waiting for the associated task(s) to finish executing [36].

**gRPC** In 2015, Google open sourced a new RPC system designed for use in modern datacenters. The gRPC documentation encourages clients to include a timeout, which it refers to as a deadline, with their requests [28]. This timeout is transmitted to the server along with the request, where it can be used to cancel the associated request. Unfortunately, each server-side function is responsible for detecting such cancellations by periodically checking a flag and responding accordingly, as shown in Listing 8.1 [29].

**ZIO gRPC** ZIO gRPC is an experimental RPC system based on gRPC. It adds asynchronous cancellation, which it accomplishes by requiring that server-side functions are written in purely functional Scala. Upon a timeout, the system transparently cancels any partial work the server has done and cleans up its associated resources [75].

## 8.2 Language support for asynchronous cancellation

Today's RPC systems force the server-side programmer to handle timeouts and cancellation manually. The only exception is ZIO gRPC, which only supports purely functional services.

This state of affairs becomes less surprising—albeit no less disappointing—if we recall that operating systems have long failed to support asynchronous cancellation (Section 2.1). Unsurprisingly, most programming languages that support shared state have struggled with the same problem.

C# used to provide the `Thread.Abort()` method for asynchronously cancelling a thread. It has since marked it obsolete because it suffers from the same safety problems as cancelling POSIX and Windows kernel threads. As the documentation remarks:

> The `Thread.Abort()` method should be used with caution. Particularly when you call it to abort a thread other than the current thread, you don't know what code has executed or failed to execute when the `ThreadAbortException` is thrown. You also

> cannot be certain of the state of your application or any application and user state that it's responsible for preserving. For example, calling `Thread.Abort()` may prevent the execution of static constructors or the release of unmanaged resources [66].

The Java standard library initially offered a `Thread.stop()` method for asynchronous cancellation, but the language was soon forced to deprecate the feature. Notably, it purports to avoid the resource leak and deadlock problems of competing systems, but can still corrupt program and library state:

> Stopping a thread causes it to unlock all the monitors that it has locked. (The monitors are unlocked as the `ThreadDeath` exception propagates up the stack.) If any of the objects previously protected by these monitors were in an inconsistent state, other threads may now view these objects in an inconsistent state. Such objects are said to be damaged. When threads operate on damaged objects, arbitrary behavior can result. This behavior may be subtle and difficult to detect, or it may be pronounced. Unlike other unchecked exceptions, `ThreadDeath` kills threads silently; thus, the user has no warning that his program may be corrupted. The corruption can manifest itself at any time after the actual damage occurs, even hours or days in the future [33].

Even scarier is Ruby, which still offers the `Thread.raise` interface and higher-level "features" such as `Timeout`. These are still marketed as useful and bear no disclaimers, despite making no effort to address any of the perils of asynchronous preemption [22].

## 8.3 Easier RPCs with *strobelight*

Our work on preemptible functions paves the way for systems that safely support both preemption and asynchronous cancellation. The new concurrency abstraction provides what the Java commentary identifies as missing: a way to determine that structures may be in a damaged state. Furthermore, it contains such damage to those objects directly used by the cancelled preemptible function by isolating library state using selective relinking. (As we have noted, resource leaks remain a problem that the programmer must solve manually, but we sketched an approach to automatic cleanup in Chapter 6.)

The *strobelight* system leverages preemptible functions to improve the usability of RPCs in two significant ways: preventing wasted work and preserving salvageable progress. It does this by extending *libturquoise*'s approach of wrapping each task in a preemptible function (Chapter 7). A *strobelight* client sends a request consisting of a function identifier, set of arguments, and its timeout. As each request arrives, the *strobelight* server invokes a preemptible function to process it. If the client times out and stops listening for a response, the server's preemptible function does so at a similar time, automatically pausing execution. The server keeps paused preemptible functions around and "memoizes" repeated requests by resuming them when an incoming function identifier and argument set match those of one that has timed out. The server also memoizes computed results, so that repeating a completed request does not result in duplicate work.

## 8.4 Future work

The *strobelight* server is a proof-of-concept system implemented in 128 lines of Rust. As such, it has several shortcomings that a production system would need to address. For one thing, it currently memoizes all partial and completed requests, assuming that all server-side functions neither affect one another's results nor depend on anything other than their arguments. Lifting these impractical restrictions would require a configuration or annotation mechanism to indicate which functions have these properties; ideally, the system would also provide transaction management that was aware of dependencies and able to determine when a restart or recalculation was required. Another shortcoming of the current system is that it keeps all past requests around forever. To avoid a ballooning resource footprint, a real server would need to use an LRU cache or similar approach. For partially-computed functions, this would either require the programmer to write a specific cleanup handler to accompany each server-side function or the integration of our automatic cancellation ideas into the preemptible functions stack. There is currently high request latency because the server spawns a kernel thread to handle each request and uses blocking I/O, but it could use a userland thread pool and futures. The current implementation does not provide type safe bindings for the client, but systems such as gRPC already solve this using Protocol Buffers. Finally, *strobelight* does not presently allow installing additional functions into a running server, but this would be easy to support thanks to *libgotcha*'s compatibility with the dynamic linker's `dlopen()` interface for runtime loading (Section 3.7.1).

## 8.5 Conclusion

This chapter introduced the *strobelight* RPC system, which uses preemptible functions to make each server-side task time out automatically when the client's request does. This saves implementers of server-side functions from having to punctuate their code with frequent deadline checks like that shown in Listing 8.1, and from relying on the authors of third-party libraries to do the same. Furthermore, the server memoizes partial computations to avoid repeated work should the client retry a failed request. The system showcases how preemptible functions empower even non-expert programmers to implement complex application functionality.

“ The ships hung in the sky in much the same way that bricks don’t. ”
— Douglas Adams, *The Hitchhiker’s Guide to the Galaxy*

# Chapter 9

# Microsecond-scale microservices

This chapter provides a case study of how lightweight preemptible functions could be used to create the serverless platform of the future. We have not prototyped any of its ideas on *libinger*; rather, they formed the initial motivation for fine-grained preemption and the inspiration for the preemptible function abstraction.

Modern cloud computing environments strive to provide users with fine-grained scheduling and accounting, as well as seamless scalability. The most recent face to this trend is the “serverless” model, in which individual functions, or microservices, are executed on demand. Popular implementations of this model, however, operate at a relatively coarse granularity, occupying resources for minutes at a time and requiring hundreds of milliseconds for a cold launch. In this chapter, we describe a novel design for providing “functions as a service” (FaaS) that attempts to be truly *micro*: cold launch times in microseconds that enable even finer-grained resource accounting and support latency-critical applications. Our proposal is to eschew much of the traditional serverless infrastructure in favor of language-based isolation. The result is microsecond-granularity launch latency, and microsecond-scale preemptive scheduling using high-precision timers.

## 9.1 Introduction

As the scope and scale of Internet services continues to grow, system designers have sought platforms that simplify scaling and deployment. Services that outgrew self-hosted servers moved to datacenter racks, then eventually to virtualized cloud hosting environments. However, this model only partially delivered two related benefits:

1. Pay for only what you use at very fine granularity

2. Scale up rapidly on demand

The VM approach suffered from relatively coarse granularity: Its atomic compute unit of machines were billed at a minimum of minutes to months. Relatively long startup times often required system designers to keep some spare capacity online to handle load spikes.

These shortcomings led cloud providers to introduce a new model, known as serverless computing, in which the customer provides *only* their code, without having to configure its environment. Such “function as a service” (FaaS) platforms are now available as AWS Lambda [2], Google

Cloud Functions [27], Azure Functions [46], and Apache OpenWhisk [5]. These platforms provide a model in which: (1) user code is invoked whenever some event occurs (e.g., an HTTP API request), runs to completion, and nominally stops running (and being billed) after it completes; and (2) there is no state preserved between separate invocations of the user code. Property (2) enables easy auto-scaling of the function as load changes.

Because these services execute within a cloud provider's infrastructure, they benefit from low-latency access to other cloud services. In fact, acting as an access-control proxy is a recurring microservice pattern: receive an API request from a user, validate it, then access a backend storage service (e.g., S3) using the service's credentials.

In this chapter, we explore a design intended to reduce the tension between two of the desiderata for cloud functions: low latency invocation and low cost. Contemporary invocation techniques exhibit high latency with a large tail; this is unsuitable for many modern distributed systems which involve high-fanout communication, sometimes performing thousands of lookups to handle each user request. Because user-visible response time often depends on the tail latency of the slowest chain of dependent responses [12], shrinking the tail is crucial [32, 73, 40, 34].

Thus we seek to reduce the invocation latency and improve predictability, a goal supported by the impressively low network latencies available in modern datacenters. For example, it now takes $< 20\mu s$ to perform an RPC between two machines in Microsoft Azure's virtual machines [23]. We believe, however, that fully leveraging this improving network performance will require reducing microservices' invocation latencies to the point where the network is once again the bottleneck.

We further hypothesize—admittedly without much proof for this chicken-and-egg scenario—that substantially reducing both the latency and cost of running intermittently-used services will enable new classes and scales of applications for cloud functions, and in the remainder of this chapter, present a design that achieves this. As Lampson noted, there is power in making systems "fast rather than general or powerful" [38], because fast building blocks can be used more widely.

Of course, a microservice is only as fast as the slowest service it relies on; however, recall that many such services are offered in the same clouds and datacenters as serverless platforms. Decreasing network latencies will push these services to respond faster as well, and new stable storage technologies such as Intel Optane (which offers sub-microsecond reads and writes) will further accelerate this trend by offering lower-latency storage.

In this chapter, we propose a restructuring of the serverless model centered around low-latency: *lightweight microservices* run in *shared processes* and are isolated primarily with language-based *compile-time guarantees* and *fine-grained preemption*.

## 9.2 Motivation

Our decision to use language-based isolation is based on two experimental findings: (1) Process-level isolation is too slow for microsecond-scale user functions. (2) Commodity CPUs support task preemption at microsecond scale. We conducted our experiments on an Intel® Xeon® E5-2683 v4 server (16 cores, 2.1 GHz) and Linux 4.13.0.[1]

[1]Source code for the benchmarks in this chapter is available from `https://github.com/efficient/microservices_microbenchmarks`.

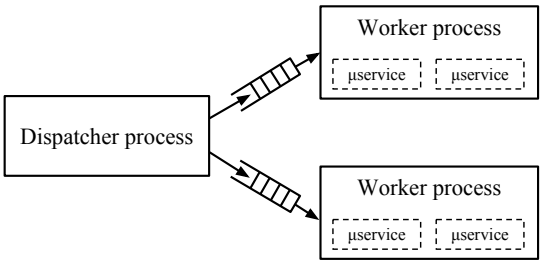



**Figure 9.1:** Language-based isolation design. The dispatcher process uses shared in-memory queues to feed requests to the worker processes, each of which runs one user-supplied microservice at a time.

### 9.2.1 Process-level isolation is too slow

We use a single-machine experiment to evaluate the invocation overhead of different isolation mechanisms: Microservices run on 14 *worker* CPU cores. Another core runs a *dispatcher* process that launches microservices on the workers. All requests originate at the dispatcher (which in a full serverless platform would forward from a cluster scheduler); it schedules ≤14 microservices at a time, one per worker core, choosing from a pool of 5,000.

To provide a comparison against contemporary system designs, we use two different isolation mechanisms:

1. **Process-based isolation:** Each microservice is a separate process. We expect this approach to exhibit latency at least as low as the container isolation common in contemporary serverless deployments.

2. **Language-based isolation:** Each worker core hosts a single-threaded *worker process* that directly executes different microservices, one at a time. In this approach, shown in Figure 9.1, a worker process runs a microservice by calling its registered function; we assume that the microservice function can be isolated from the worker process with language-based isolation techniques that we discuss in Section 9.3. The dispatcher schedules microservices on worker processes by sending them requests on a shared memory queue, which idle worker processes poll.

We use 5,000 copies of a Rust microservice that simply records a timestamp: latency is measured between when the dispatcher invokes a microservice and the time that microservice records. There are two experiment modes:

**Warm-start requests.** We first model a situation where all of the microservices are already resident on the compute node. In the case of process-based isolation, the dispatcher launches all 5,000 microservices at the beginning of the experiment, but they all block on an IPC call; the dispatcher then invokes each microservice by waking up its process using a UDP datagram. In the case of language-based isolation, the microservices are dynamic libraries preloaded into the worker processes.

Table 9.1 shows the latency and throughput of the two methods. We find that the process-based isolation approach takes 9 µs and achieves only 300,000 warm microservice invocations

| Microservices | | Latency (µs) | | Throughput |
|---|---|---|---|---|
| Resident? | Isolation | Median | 99% | (M invoc/s) |
| Warm-start | Process | 8.7 | 27.3 | 0.29 |
| | Language | 1.2 | 2.0 | 5.4 |
| Cold-start | Process | 2845.8 | 15976.0 | – |
| | Language | 38.7 | 42.2 | – |

**Table 9.1:** Microservice invocation performance

per second. In contrast, language-based isolation achieves 1.2 µs latency (with a tail of just 2.0 µs) and over 5 million invocations per second.

Considering that the FaRM distributed computing platform achieved mean TATP transaction commit latencies as low as 19 µs in 2015 [16], a 9 µs microservice invocation delay represents almost 50% overhead for a microservice providing a thin API gateway to such a backend. We therefore conclude that even in the average case, process-based isolation is too slow for microsecond-scale scheduling. Furthermore, IPC overhead limits invocation throughput.

Process-based isolation also has a higher memory footprint: loading the 5,000 trivial microservices consumes 2 GiB of memory with the process-based approach, but only 1.3 GiB with the language-based one. However, this benefit may reduce as microservices' code sizes increase.

**Cold-start requests.** Achieving ideal wakeup times is possible only when the microservices are already resident, but the tail latency of the serverless platform depends on those requests whose microservices must be loaded before they can be invoked. To assess the difference between process-based and language-based isolation in this context, we run the experiment with the following change: In the former case, the dispatcher now launches a transient microservice process for each request by `fork()/exec()`'ing. In the latter, the dispatcher asks a worker to load a microservice's dynamic library (and unload it afterward). The results in Table 9.1 reveal an order-of-magnitude slip in the language-based approach's latency; however, this is overshadowed by the three orders of magnitude increase for process-based isolation.

### 9.2.2 Intra-process preemption is fast

In a complete serverless platform, some cluster-level scheduler would route incoming requests to individual worker nodes. Since we run user-provided microservices directly in worker processes, a rogue long-running microservice could thwart such scheduling by unexpectedly consuming the resources of a worker that already had numerous other requests queued. We hypothesize that, in such situations, it is better for tail latency to preempt the long microservice than retarget the waiting jobs to other nodes in real time. (Only the compute node already assigned a request is well positioned to know whether that request is being excessively delayed: whereas other nodes can only tell that the request hasn't yet *completed*, this node alone knows whether it has been *scheduled*.) At our scale, this means a preemption interval up to two orders of magnitude faster than Linux's default 4 ms process scheduling quantum. Section 5.6.1 demonstrates that interval timers are capable of delivering signals with this frequency.

## 9.3 Providing isolation

Consolidating multiple users' jobs into a single process requires addressing security and isolation. We aim to do it without compromising our ambitious performance goals.

Our guiding philosophy for achieving this is "language-based isolation with defense in depth." We draw inspiration from two recently-published systems whose own demanding performance requirements drove them to perform similar coalescing of traditionally independent components: NetBricks [50] is a network functions runtime for providing programmable network capabilities; it is unique among this class of systems for running third-party network functions in-process rather than in VMs. Tock [39] is an embedded microkernel whose servers ("capsules") form a common compilation unit and communicate using type-safe function calls. As their primary defense against untrusted code, both systems leverage Rust [54], a new type-safe systems programming language.

Rust is a strongly-typed, compiled language that uses a lightweight runtime similar to C. Unlike many other modern systems languages, Rust is an attractive choice for predictable performance because it does not use a garbage collector. It provides strong memory safety guarantees by focusing on "zero-cost abstractions" (i.e., those that can be compiled down to code whose safety is assured without runtime checks). In particular, safe Rust code is guaranteed to be free of null or dangling pointer dereferences, invalid variable values (e.g., casts are checked and unions are tagged), reads from uninitialized memory, mutations of non-`mut` data (roughly the equivalent of C's `const`), and data races, among other misbehaviors [56].

We require each microservice to be written in Rust (although, in the future, it might be possible to support subsets of other languages by compiling them to safe Rust), giving us many aspects of the isolation we need. It is difficult for microservices to crash the worker process, since most segmentation faults are prevented, and runtime errors such as integer overflow generate Rust panics that we can catch. Microservices cannot get references to data that does not belong to them thanks to the variable and pointer initialization rules.

Given our performance goals, there is a crucial isolation aspect that Rust does not provide: there is nothing to stop users from monopolizing the CPU. Our system, however, must be preemptive. We can apply preemptible functions (Chapter 2) here to impose a time quota on each microservice. This has the added benefit of automatically providing memory isolation between library dependencies (Chapter 3), insulating unsafe platform code from affecting the state of other microservices or the rest of the worker process.

Our defense-in-depth comes from using lightweight operating-system protections to block access to certain system calls, as well as the proposed mechanisms in Section 9.5. Some system calls must be blocked to have any defense at all; otherwise, the microservice could create kernel threads (e.g., `fork()`), create competition between threads (e.g., `nice()`), or even terminate the entire worker (e.g., `exit()`). Finally, user functions should not have unmonitored file system access.

We propose to block system calls using Linux's `seccomp()` system call [58]; each worker process should call this during initialization to limit itself to a allowlisted set of system calls. Prior to lockdown, the worker process should install a `SIGSYS` handler for regaining control from any microservice that attempts to violate the policy.

## 9.4 Deployment

We now describe our microservices in the broader context of our full proposed serverless system. We clarify their lifecycle, interactions with the compute nodes, and the trust model for the cloud provider.

Users submit their microservices in the form of Rust source code, allowing the serverless operator to pass the `-Funsafe-code` compilation flag to reject any `unsafe` code. This process need not occur on the compute nodes, provided the deployment server tasked with compilation runs the same version of the Rust compiler.[2] The operator needs to trust the compiler, standard library, and any libraries against which it will permit the microservice to link (since they might contain `unsafe` code), but importantly need not worry about the microservice itself.

We believe that restricting microservices to a specific list of permitted dependencies is reasonable. Any library that contains only safe Rust code could be allowlisted without review. To approximate the size of such a list given the current Rust ecosystem, we turn to a 2017 study [9] by the Tock authors that found just under half of the Rust package manager's top 1000 most-downloaded libraries to be free of unsafe code. They caution that many of those packages have unsafe dependencies, but reviewing a relatively small number of popular libraries would open up the majority of the most popular packages.

If the application compiles (is proven memory-safe) and links (depends only on trusted libraries) successfully, the deployment server produces a shared object file, which the provider then distributes to each compute node on which it might run. Then, in order to ensure that invokers will experience the warm-start latencies discussed in Section 9.2, those nodes' dispatcher processes should instruct one or more of their workers to preload the dynamic library. If the provider experiences too many active microservices for its available resources, it can unload some libraries; on their next invocation, they will experience higher (cold start) invocation latencies as they synchronously load the dynamic library.

## 9.5 Future work

As noted above, our exploration is preliminary; this section outlines several open questions. These questions fall into two categories: simplifications we made for benchmarking and defense-in-depth safeguards against unexpected failures (e.g., compiler bug or the operator allowing use of a buggy or malicious library).

**Host process.** Our benchmarks do not account for isolation between the dispatcher and worker processes. A real deployment would want to employ standard OS techniques to reduce the chance of interference by a misbehaving worker. Examples include auditing interactions with the shared memory region to ensure invalid or inconsistent data originating from a worker cannot create an unrecoverable dispatcher error; handling the `SIGCHLD` signal to detect a worker that has somehow crashed; and keeping a recovery log in the dispatcher process so that any user jobs lost to a failed worker process can be reassigned to operational workers.

[2] This restriction exists because, as of the latest release (1.23.0) of the compiler, Rust does not have a stable ABI.

**Further defense in depth with ERIM.** ERIM outlines a set of techniques and binary rewriting tools useful for using Intel's Memory Protection Keys to restrict memory access by threads within a process [70]. While preliminary and without source yet available, this appears to be an attractive approach for defense-in-depth both within worker processes and between the workers and the dispatcher.

**Library functions.** As with system calls, there may exist library functions in Rust (and certainly in libc, which we deny by default) that are unsafe for microservices to access. Because the Rust standard library requires unsafe code, defense-in-depth suggests that a allowlisting-based approach should be employed for access to its functions. Certainly library functions must be masked—for example, our use of Rust's panic handler for preemption means that we must deny microservice code the ability to catch the panic and return to execution. Although we mitigate this possibility by detecting and blocklisting microservices that fail to yield under a SIGALRM, it would be desirable to block such behavior entirely. Possible options include using the dynamic linker to interpose stub implementations or linking against a custom build of the library, or using more in-depth static analysis.

**Resource leaks.** Safe Rust code provides memory safety, but it cannot prevent memory leaks [57]. For example, destructor invocation is not guaranteed using Rust's default reference counting-based reclamation; therefore, unwinding the stack during preemption is not guaranteed to free all of a microservice's memory or other resources. Potential solutions are interposing on the dynamic allocator to record tracking information (likely proving expensive) or using per-microservice heaps that main worker process can simply deallocate when terminating a microservice. The worker can also deallocate other resources, such as unclosed file descriptors. If these checks end up being too expensive, the worker could execute its cleanup after a certain number of microservices have run or when the load is sufficiently low.

**Side channels.** Our current approach is vulnerable to side-channel attacks [45, 37]. For example, microservices have access to the memory addresses and timings of dynamic memory allocations, as well as the numbers of opened file descriptors. Although side-channels exist in many systems, the short duration of microservice functions may make mounting such attacks more challenging; nevertheless, standard preventative practices found in the literature should apply.

Despite the security challenges of running microservice as functions, worker processes are still well-isolated from the rest of the system. Worst case, the central dispatcher process can restart a failed worker and automatically ban suspect microservices.

## 9.6 Conclusion

In order to permit applications to fully leverage the 10s of $\mu s$ latencies available from the latest datacenter networks, we propose a novel design for serverless platforms that runs user-submitted microservices within shared processes. This structure is possible because of language-based *compile-time memory safety guarantees* and *microsecond-scale preemption*. Our benchmark demonstrates that these goals of high throughput, low invocation latency, and rapid preemption

are achievable on today's commodity systems, while potentially supporting hundreds of thousands of concurrently available microservices on each compute node. We believe that these two building blocks will enable new FaaS platforms that can deliver single-digit microsecond invocation latencies for lightweight, short-lived tasks.

Since we published this benchmark, Cloudflare has built and deployed a production FaaS platform called Workers. Like our proposed architecture, this system omits containers and virtual machines by running user code in process [11]. It accomplishes this by requiring users to submit JavaScript code (or WebAssembly) and running each task as a separate Isolate under the V8 JavaScript engine [69], thereby halving major competitors' cold-start latencies. While running under a JavaScript engine confers some practical benefits such as not having to audit dependencies, we believe that a shift to native code will be necessary to further reduce cold-start latencies from the millisecond range to the tens or hundreds of microseconds.

# Chapter 10

# Conclusions and continuations

The dissertation set out to substantiate this thesis statement:

> Providing language-agnostic abstractions for fine-grained preemption and function-level isolation enables the straightforward implementation of application functionality long considered prohibitively difficult, such as preemptive user threads and asynchronous task cancellation.

We now break this down and briefly recap our work pertaining to each of its claims (Section 10.1), discuss applications and directions for future work (Section 10.2), review a selection of the technical challenges we had to overcome (Section 10.3), and distill a few lessons for future systems builders (Section 10.4).

## 10.1 Contributions

The abstract expands on the thesis statement:

> We introduce novel programming abstractions for isolation of both time and memory. They operate at finer granularity than traditional primitives, supporting preemption at sub-millisecond timescales and tasks defined at the level of a function call. This resolution enables new functionality for application programmers, including users of unmanaged systems programming languages, all without requiring changes to the existing systems stack. Despite being concurrency abstractions, they employ synchronous invocation to allow application programmers to make their own scheduling decisions. However, we found that they compose naturally with existing concurrency abstractions centered around asynchronous background work, such as threads and futures. We demonstrated how such composition can enable asynchronous cancellation of threads and the implementation of preemptive thread libraries in userland, both regarded for decades as challenging problems.

It makes specific references to key contributions; we now further expand on each of these and give pointers back to the relevant chapters and sections.

**Novel programming abstractions for isolation of both time and memory** These abstractions are lightweight preemptible functions and selective relinking. They are introduced in Chapter 2 and Chapter 3, respectively.

**Preemption at sub-millisecond timescales** We found in Section 5.6.1 that the modern systems stack is capable of supporting timer signals with periods on the order of microseconds. We argued that the design of lightweight preemptible functions is compatible with preemption quanta down to at least the tens of microseconds, with scaling limited by increasing CPU time overheads. We also demonstrated that the latency of invoking a preemptible function is in the same order of magnitude (Section 5.12). This is one order of magnitude faster than forking a new process and two orders of magnitude finer than the typical operating system preemption quantum.

**Tasks defined at the level of a function call** Both lightweight preemptible functions and selective relinking explicitly treat function calls as an isolation boundary. The former expresses this boundary by asking the programmer to annotate preemptible functions by invoking them with a wrapper function (Section 2.3); the latter does so by intercepting calls to dynamically-linked functions based on the context of the call (Section 3.3) and knowledge of specific functions that cannot be protected solely through memory isolation (Section 3.6). That our system understands function calls is significant because traditionally both preemption and memory isolation have operated exclusively at the granularity of a kernel thread.

**New functionality for application programmers** It is worth emphasizing that our abstractions are available in userland and accessible to any programmer experienced with concurrency. They expose powerful APIs that we summarize in Listings 5.1 and 5.2 (lightweight preemptible functions) and Listings 3.2 and 3.3 (selective relinking). The preemptible functions Rust API is even type safe and usable in contexts where the standard library does not allow thread spawns (Chapter 5). Preemptible functions themselves serve as an example application for selective relinking, and Chapters 7, 8, and 9 present case studies in building systems atop preemptible functions. Simpler applications include detection of pathological cases such as adversarially-constructed compressed images (Section 5.12.1) and graceful degradation by dropping video frames.

**Unmanaged systems programming languages** Unlike prior art, our abstractions are restricted neither to purely functional code nor to managed languages with heavyweight, garbage-collected runtimes (Section 2.2). In principle, their only operating system and runtime requirements are timer signals (Section 5.6) and dynamic linking (Section 3.8.1).[1] We officially support the low-level C and Rust systems programming languages. We have tried to keep the API language agnostic, and the fact that many languages include C foreign-function interfaces means that some of them may already be able to use preemptible functions out of the box.

[1] It helps to also have exception handling, as per Section 6.2.

**Without requiring changes to the existing systems stack** We implement everything entirely in userland by building on existing abstractions such as dynamic linking, memory protection, POSIX signals and timers, and exception handling. Where we alter the behavior of the dynamic linker and C runtime, we make those changes at load time (Section 3.4). (While we do not in principle require a custom glibc, one must rebuild from source with a different configuration macro for full functionality, as explained in Section 3.8.2. The only component of the system that absolutely requires the developer of an application to rebuild its dependencies is cancellation resource cleanup, per Sections 6.4.5, 6.4.7, and 6.5.)

**Synchronous invocation** Unlike threads and callback-based futures, preemptible functions are invoked synchronously. This eliminates the need for an external scheduler and the associated overhead in cases where the programmer is willing to manually manage which preemptible function to launch or resume next (Chapter 2). It also allows a preemptible closure to safely capture local variables in Rust (Chapter 5).

**Compose naturally with existing concurrency abstractions** Both abstractions are usable in multithreaded contexts. In fact, one can even take a paused preemptible function that was executing one one kernel thread and resume it on a different one (Section 5.3), a property that allows schedulers to treat preemptible functions like any other task. Preemptible functions can be used to construct preemptible futures (Section 7.2) and even mutexes with `await`-style call with current continuation semantics (Section 5.1.1).

**Asynchronous cancellation of threads** It has long proven difficult to cancel running threads, both at the operating system level (Section 2.1) and at the language level (Section 8.2). We discover that we can leverage the memory isolation provided by selective relinking to enable asynchronous cancellation of POSIX threads, a feature that is almost unusable as shipped in the Unix operating system and its clones (Chapter 4.2). Although resource cleanup requires careful programmer attention (Section 4.2.1), we argue that this is possible to address automatically, even without garbage collection, for languages conforming to RAII (Chapter 6).

**Preemptive thread libraries in userland** By modifying the thread pool from a Rust futures executor to transparently wrap tasks in preemptible futures, we have created arguably the first general-purpose preemptive thread library implemented entirely in userland. Details are in Chapter 7.

## 10.2 Applications and future work

This dissertation leaves ample opportunity for future work on lightweight preemptible functions and selective relinking. Possible directions include exploring other applications for our abstractions, conducting a deeper investigation of our example applications, making performance improvements, lifting scalability restrictions, adding defense in depth, improving application compatibility, and contributing more of our discoveries upstream.

There are plenty of possible applications we have not explored. We have focused on preemptible functions that use timeouts as a resource limit, but the underlying fine-grained preemption we developed to support them is actually more general. One can imagine applying it to real-time scheduling or imposing quotas based on resources other than time, such as data transferred or number of page faults. Chapter 9 proposed the use of preemptible functions to merge multiple cloud tenants' microservices into a single worker process, a technique that could be applied to locally-running programs as well. One could even write a tool that took two or more dynamically-linked position-independent executables and merged them into a single application that included in-process scheduling. Selective relinking, with its ability to intercept function calls and issue notification callbacks, surely has applications in aspect-oriented programming. Its techniques could be applied to other problems as well, such as allowing applications to depend on more than one version of a dynamic library. Both abstractions enable numerous new use cases, and we are sure there are many we have not thought of.

Those applications that we have explored also merit deeper study. Section 4.1 demonstrated how selective relinking can be used to lift the primary safety restriction on signal handlers. Our treatment was brief, but in our opinion it represents such a significant improvement to the signal abstraction that it would be worth carrying beyond the prototype stage. In creating an implementation suitable for deployment, one might explore isolating each signal handler from the others or defining a completely safe interface for signal handling in Rust or another language with a sound type system. Section 4.2 gave a preliminary implementation of asynchronous thread cancellation, a feature that operating systems and programming languages alike have long struggled to support. Refining this prototype might involve fixing the resource leaks problem by integrating the automatic cleanup approach we sketched in Chapter 6. We think that preemptible futures and preemptive userland threading hold enormous potential for building scalable systems with better resistance to denial of service attacks. We implemented these concepts before we had proper support for thread-local storage and at a time when the Rust ecosystem was in flux because the language was just stabilizing futures and async/await. As such, the majority of the preemptible future code is compatibility calls to convert between different futures interfaces, and the thread pool works only with a very old version of the Tokio futures executor. Porting to a modern futures executor and leveraging *libinger*'s support for thread locals would permit experiments on the latest high-performance systems.

The performance of our implementation could be improved in several ways. As Section 5.6.1 noted, one current limitation is our use of a globally constant preemption quantum. We could reduce the throughput overhead while preserving preemption granularity by varying the interval based on the requested timeout and delaying the first signal for longer-running preemptible functions. Even more granular preemption might be achievable by using hardware interrupts directly instead of paying the overhead of POSIX signals; options include a custom kernel module or porting to the Dune system [7]. We saw in Section 3.9.3 that the increased TLS size has an impact on thread spawn performance. Incorporating a more robust implementation of the workaround we prototyped in that section into *libgotcha* would mitigate much of this effect. This might also eliminate the need to preallocate TCBs up front to keep preemptible function launch latencies low.

Another area for improvement is scalability, as ours is currently constrained in multiple ways. Preemptible functions' stacks have a fixed size, but leveraging demand paging would resolve this problem and also avoid having to preallocate them (Section 5.5). The fact that we need a dedi-

cated preemption signal for each preemptible function places a fixed upper bound on parallelism, but (mis)using glibc's nonstandard `SIGEV_THREAD_ID` feature intended for thread libraries could make a single signal sufficient (Section 5.6). That the dynamic linker supports a limited number of namespaces determined at compile time places a fixed upper bound on concurrency, but one could port selective relinking to an alternative dynamic linker such as drow (Section 3.8.2). Our current implementation reduces runtime latency at the cost of startup time, which is fine for long-running processes or where workers can be spawned from template "zygote" processes, but could pose issues otherwise (Section 3.9.2). The performance and scalability improvements proposed thus far are likely to make pool allocators unnecessary for TCBs and stacks, the largest resources we preallocate. Our remaining startup overhead comes from preparing all libsets at load time, a tradeoff that one would already have to revisit in order to support a variable number of libsets.

Applying our isolation mechanisms to multi-tenancy situations as proposed in Chapter 9 would require defense in depth. Both *libgotcha* and *libinger* would benefit from using enforced interposition (Section 3.7.1) to replace library functions that could be used to circumvent selective relinking and preemption in isolated code regions (e.g., by interfering with signals). One might also consider expanding the preemptible functions interface to allow configurable isolation of other actions that could affect the rest of the application, such as raising exceptions (Section 5.9).

There are a few enhancements that would improve compatibility, making more unmodified existing code work with libsets and within preemptible functions. One could replace library functions that exhibit unusual signal interruption behavior with wrappers that hid those differences (Section 5.10). One could detect signals already used by the application to avoid conflicting allocation of the same signals for preemption (Section 5.6). One could improve interception of global variable accesses to properly support symbols of any size and remove the reliance on heuristics that cannot handle certain instruction sequences (Section 3.5.2). One could consider supporting preemptible functions that spawned threads and/or forked new processes. Finally, one could implement support for nested preemptible functions.

While we have discovered and reported multiple bugs over the course of this project, we have encountered other issues that may or may not be worth addressing upstream. In some cases, it was unclear whether issues truly represented a misimplementation of the relevant specification, or whether they were relevant outside our own specific and arcane use of runtime features. It is likely worth revisiting our workarounds and considering which could be reimplemented upstream to benefit other users. Examples include `_Unwind_RaiseException()`'s linker namespace limitations (Section 3.7.1), GDB's reluctance to load symbols from modules loaded in alternate namespaces (Section 3.8.1), scaling problems in glibc's allocation of TCBs (Section 3.9.3), certain glibc functions misbehaving in alternate namespaces (Section 5.10), and what may be an off-by-one error in libgcc and libunwind's implementation of the DWARF specification (Section 6.4.7).

## 10.3 Technical challenges

Manipulating GOTs is not the only thing that gives *libgotcha* its name. Over the course of this project, we encountered and had to adapt to a number of tricky low-level details of other systems. We also developed some possibly novel insights and had to create some hackery of our own.

We spent a lot of time comprehending and responding to implementation details of glibc, and especially its dynamic linker. One of the early setbacks was its use of `NODELETE` shared libraries (Section 3.4) that monkey patch one another's internal state (Section 3.6.1). Debugging unexpected crashes while calling some library functions, we learned about the nonstandard `GNU_IFUNC` relocation type that we had not accounted for because of its absence from the ABI specification (Section 3.5.1). Once we had more complex programs running, we were surprised to find that exceptions were causing Rust programs to abort, even though we had built with exception unwinding and were careful not to allow exceptions to propagate across foreign function call boundaries (Section 5.9). The cause turned out to be poor interplay between the stack unwind library and the dynamic linker's `_dl_iterate_phdr()` interface in the presence of calls between linker namespaces (Section 3.7.1). When we started to run programs with multiple libsets, they crashed and we learned about the dynamic library's static TLS surplus (Section 3.8.2); at the time, tweaking this required modifying a macro in the dynamic linker sources, but it has since been moved to a tunable configured via an environment variable. This is not the only recent upstream change to affect us (Section 3.8.1). When we added support for thread-local storage, we found that in order to allocate our own TCBs without creating POSIX threads, we needed to manually initialize some critical fields (Section 5.3). This led to another issue that briefly broke threads' ability to signal themselves (Section 3.7.2). Finally, when writing *libas-safe*, we had to grok libc's initialization code in order to inject a libset switch late enough in startup that it did not leave the starting libset's C runtime broken (Section 4.1).

We also encountered a few dark corners of the Rust language. Although Rust's ABI is technically unstable, it mostly follows the C ABI. However, there are exceptions, and the lack of stability means there is no single reference of the deviations. After hours debugging mysterious crashes, we learned the hard way about one such difference: Rust uses `%rdx` as a second return register to store the other half of fat pointers. We had to tweak our trampoline code to preserve this register on return callbacks following uninterruptible code (Section 3.6.2). Once we started implementing preemptible functions, we quickly ran into double frees caused by the interaction between POSIX contexts and destructors. We also stumbled upon a self-referential pointer in the definition of the POSIX context structure (Section 5.4.1). This can lead to undefined behavior if contexts are not heap allocated or when we need to copy one over another (Section 5.7). In such cases, we manually update the troublesome pointer just before restoring the context.

Implementing *libgotcha* required writing some elaborate low-level code. To reroute dynamic function calls based on runtime information, we had to inject code at the start of the call. We had to write this code in assembly to avoid clobbering registers, and design our data structures to be easy to access without compiler-generated code (Section 3.5.1). This also meant storing thread-local variables using a TLS model that did not require making calls into the runtime in order to resolve addresses (Section 3.7.2). To disambiguate which function was being called, we had to generate executable pages of stub functions based on a common template, but with slight differences between each entry (Section 3.5.1). Supporting control library callbacks required pushing a trampoline onto the stack to mark the call that had prompted the transition into uninterruptible code (Section 3.6.2). We had to size the trampoline's stack frame to maintain stack alignment, and incorporate logic to prevent any libset switches inside the callback from prompting recursive callbacks. Intercepting global variable accesses was challenging in a different way, requiring a difficult-to-debug segmentation fault handler, a dependency on a disassembler library, and custom heuristics based on patterns we observed in compiled code (Section 3.5.2). We wanted enforced

interposition functions to be able to call the functions they wrapped without manually resolving the underlying symbol (Section 3.7.1). Preventing GCC and Clang from assuming these calls were recursive required a combination of the `-fno-optimize-sibling-calls` compiler switch and symbol aliases.

Finally, we had to develop some insights of our own that might prove useful to others working with signals, POSIX contexts, exceptions, or dynamic linking. We created a portable mechanism for directing external signals at particular threads using a signal pool and convergence algorithm (Section 5.6). We also discovered a way of safely restoring a POSIX context obtained from a signal handler (Section 5.7.1). In our exploration of automatic resource cleanup, we sketched out workarounds to some of the problems that have long plagued asynchronous exception handling (Section 6.4). We designed algorithms for detecting cross-module dynamic symbol references (Section 3.3.1) and lazily reinitializing portions of the TLS area on demand (Section 3.7.2). In extending pointer equality to selective relinking, we realized that two pointers can be statically equivalent within a certain context even without knowing which libset their calls will be routed to at runtime (Section 3.5.1). In the latter section, we also invented a trick for convincing the dynamic linker to write the address of lazily-resolved symbol addresses to a custom location.

## 10.4 Lessons for systems builders

We leave the reader with a few higher-level lessons that have saved us time and pain, but would have saved us more of each if we had fully appreciated them from the start:

- You cannot have fine-grained time isolation without fine-grained memory isolation.
- Design abstractions modularly and with an eye to other use cases; for instance, do not try to fold time and memory isolation into a single tightly-coupled primitive as we almost did.
- The ability to resume interrupted tasks is a useful feature that improves composability and is cheaper than cancellation, but it also introduces explicit concurrency into the task interface.
- Treat debuggability as a first-order concern. Include runtime assertions with descriptive errors, trace safety violations in unsound interfaces at runtime in debug builds, detect and warn on misuse that could cause confusing or invalid results, make it easy to disable complex features that may create or obscure problems, and test and maintain support for running under debugging and diagnostic tools.
- Periodically teach students and peers about aspects of your system. Often you (or they) will discover design shortcomings, interface paper cuts, or pivotal enhancements. Even if not, you will come to better understand its workings and be able to clearly articulate its insights and their broader implications.
- Systems programming languages could save many programmers from handwriting assembly code by providing a portable way to specify that a function should preserve all of its caller's registers. GCC and LLVM provide the `no_caller_saved_registers` function attribute, but it is only implemented for the x86 family of architectures and its effect on non–general-purpose registers differs between these two code generators.

# Bibliography


[1] *Proc. 16th USENIX NSDI*, Boston, MA, Feb. 2019.

[2] Amazon. AWS Lambda. `https://aws.amazon.com/lambda`.

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the thirteenth ACM symposium on operating system principles (SOSP '91)*, 1991.

[4] Android Linker Namespace Bypass Library. `https://github.com/bylaws/liblinkernsbypass`, 2021.

[5] Apache Software Foundation. OpenWhisk. `https://openwhisk.apache.org`.

[6] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe. A fork() in the road. In *HotOS '19: Proceedings of the workshop on hot topics in operating systems*, May 2019.

[7] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazères, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementaiton (OSDI'12)*, 2012.

[8] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky. Lightweight preemptible functions, 2020.

[9] Brad Campbell. Crates.io ecosystem not ready for embedded Rust. `https://www.tockos.org/blog/2017/crates-are-not-safe`.

[10] A. Clements. Go runtime: tight loops should be preemptible. `https://github.com/golang/go/issues/10958`, 2015.

[11] CloudFlare. Cloud computing without containers. `https://blog.cloudflare.com/cloud-computing-without-containers`.

[12] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb. 2013.

[13] T. Delisle. Concurrency in C∀. `https://uwspace.uwaterloo.ca/handle/10012/12888`, 2018.

[14] Deno: A modern runtime for JavaScript and TypeScript. `http://deno.land`.

[15] dlmopen(3) manual page from Linux man-pages project, 2019.

[16] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[17] drow-loader. `https://github.com/jtracey/drow-loader`, 2022.

[18] DWARF debugging information format version 5. `https://dwarfstd.org/doc/DWARF5.pdf`, 2017.

[19] M. J. Eager. Introduction to the DWARF debugging format. Technical report, 2012.

[20] D. Eloff. Go proposal: a faster C-call mechanism for non-blocking C functions. `https://github.com/golang/go/issues/16051`, 2016.

[21] G. H. et al. An overview of the Singularity project. Technical Report MSR-TR-2005-135, Microsoft Research Technical Reports, 2005.

[22] J. Evans. Why Ruby's Timeout is dangerous (and Thread.raise is terrifying). `https://jvns.ca/blog/2015/11/27/why-rubys-timeout-is-dangerous-and-thread-dot-raise-is-terrifying`, 2015.

[23] D. Firestone et al. Azure accelerated networking: SmartNICs in the public cloud. In *Proc. 15th USENIX NSDI*, Renton, WA, Apr. 2018.

[24] getcontext(3) manual page from Linux man-pages project, Oct. 2019.

[25] The Go programming language. `https://golang.org`, 2019.

[26] Go 14 release notes. `https://go.dev/doc/go1.14`, 2020.

[27] Google. Cloud Functions. `https://cloud.google.com/functions`.

[28] gRPC, a high performance, open-source universal RPC framework. `http://www.grpc.io/`, 2017.

[29] gRPC and deadlines. `http://grpc.io/blog/deadlines`, 2018.

[30] C. T. Haynes and D. P. Friedman. Engines build process abstractions. Technical Report TR159, Indiana University Computer Science Technical Reports, 1984.

[31] hyper: Fast and safe HTTP for the Rust language. `https://hyper.rs`, 2019.

[32] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[33] Java thread primitive deprecation. `https://docs.oracle.com/javase/6/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html`.

[34] M. Jeon, Y. He, H. Kim, S. Elnikety, S. Rixner, and A. L. Cox. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[35] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for microsecond-scale tail latency. In *Proc. 16th USENIX NSDI* [1].

[36] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter RPCs can be general and fast. In *Proc. 16th USENIX NSDI* [1].

[37] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[38] B. W. Lampson. Hints for computer system design. In *Proceedings of the ninth ACM symposium on operating systems principles*, SOSP '83, pages 33–48. ACM, 1983.

[39] A. Levy, B. Campbell, B. Ghena, D. B. Giffin, P. Pannuto, P. Dutta, and P. Levis. Multiprogramming a 64 kB computer safely and efficiently. In *Proceedings of the 26th ACM symposium on operating systems principles*, SOSP '17, pages 234–251. ACM, 2017.

[40] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[41] libev. `http://libev.schmorp.de`.

[42] libevent. `https://libevent.org`.

[43] libuv: Cross-platform asynchronous I/O. `https://libuv.org`.

[44] __errno_location. `http://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic/baselib---errno-location.html`, 2015.

[45] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[46] Microsoft. Azure Functions. `https://azure.microsoft.com/services/functions`.

[47] M. S. Mollison and J. H. Anderson. Bringing theory into practice: A userspace library for multicore real-time scheduling. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Philadelphia, PA, 2013.

[48] mordor: A high-performance I/O library based on fibers. `https://github.com/mozy/mordor`.

[49] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush. Engineering record and replay for deployability. Technical Report arXiv:1705.05937, arXiv, 2017.

[50] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. 12th USENIX OSDI*, Savannah, GA, Nov. 2016.

[51] pthread_setcanceltype() manual page from Linux man-pages project, 2017.

[52] G. Randers-Pehrson. Defending libpng applications against decompression bombs. `https://libpng.sourceforge.io/decompression_bombs.html`, 2010.

[53] RPC: Remote Procedure Call protocol specification version 2. `https://datatracker.ietf.org/doc/html/rfc5531`.

[54] The Rust programming language. `https://www.rust-lang.org`, 2019.

[55] The Rust programming language: Extensible concurrency with the sync and send traits. `https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html`, 2019.

[56] The Rust reference: Behavior considered undefined. `https://doc.rust-lang.org/stable/reference/behavior-considered-undefined.html`, 2018.

[57] The Rust reference: Behavior not considered unsafe. `https://doc.rust-lang.org/stable/reference/behavior-not-considered-unsafe.html`, 2019.

[58] seccomp(2) manual page from Linux man-pages project, Nov. 2017.

[59] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference*, Boston, MA, 2012.

[60] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *2005 USENIX Annual Technical Conference*, Anaheim, CA, 2005.

[61] sigaction(2) manual page from the Linux man-pages project, 2019.

[62] signal(7) manual page from the Linux man-pages project, 2019.

[63] signal-safety(7) manual page from Linux man-pages project, 2019.

[64] E. Stepanov and K. Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Francisco, CA, 2015.

[65] TerminateThread function. `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-terminatethread`, 2018.

[66] Thread.Abort method (System.Threading). `https://docs.microsoft.com/en-us/dotnet/api/system.threading.thread.abort?view=net-6.0`.

[67] timer_create(2) manual page from the Linux man-pages project, May 2020.

[68] Tokio thread pool. `https://github.com/tokio-rs/tokio/tree/tokio-threadpool-0.1.16/tokio-threadpool`, 2019.

[69] V8 JavaScript engine. `http://v8.dev`.

[70] A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, and P. Druschel. Erim: Secure and efficient in-process isolation with memory protection keys. *arXiv preprint arXiv:1801.06822*, 2018.

[71] C. J. Vanderwaart. Static enforcement of timing policies using code certification. Technical Report CMU-CS-06-143, Carnegie Mellon Computer Science Technical Report Collection, 2006.

[72] wrk: Modern HTTP benchmarking tool. `https://github.com/wg/wrk`, 2019.

[73] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[74] M. Zalewski. Delivering signals for fun and profit. `https://lcamtuf.coredump.cx/signals.txt`.

[75] ZIO gRPC: Write gRPC services and clients with ZIO. `https://scalapb.github.io/zio-grpc`, 2020.