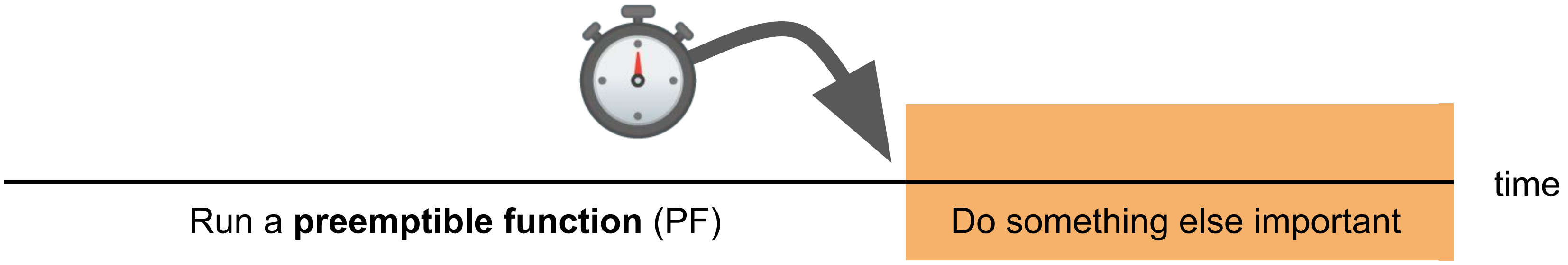
Lightweight Preemptible Functions

Sol Boucher, *Carnegie Mellon University*

Joint work with:
Anuj Kalia, *Microsoft Research*
David G. Andersen, *CMU*
Michael Kaminsky, *BrdgAI/CMU*

Light·weight (adj.): Low overhead, cheap

Pre·empt·i·ble (adj.): Able to be stopped



Why?

- Bound resource use
- Balance load of different tasks
- Meet a deadline (e.g., real time)

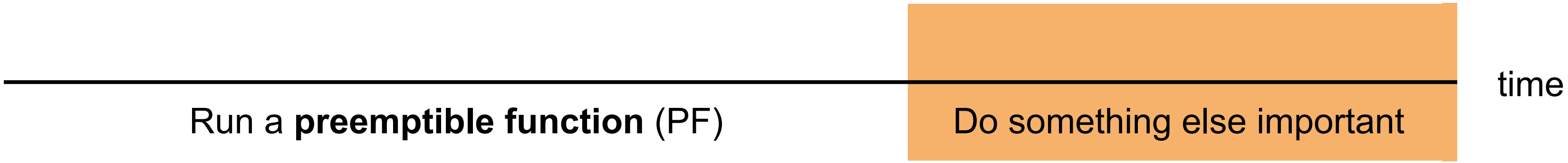
Desiderata

- Retain programmer’s control over the CPU
- Be able to interrupt arbitrary unmodified code
- Introduce minimal overhead in the common case
- Support cancellation
- Maintain compatibility with the existing systems stack

Agenda

- **Why contemporary approaches are insufficient**
  - Futures
  - Threads
  - Processes
- Function calls with timeouts
- Backwards compatibility
- Preemptive userland threading

Problem: calling a function cedes control




```
func()
```

Two approaches to multitasking

cooperative vs. **preemptive**

≈

lightweightness vs. generality

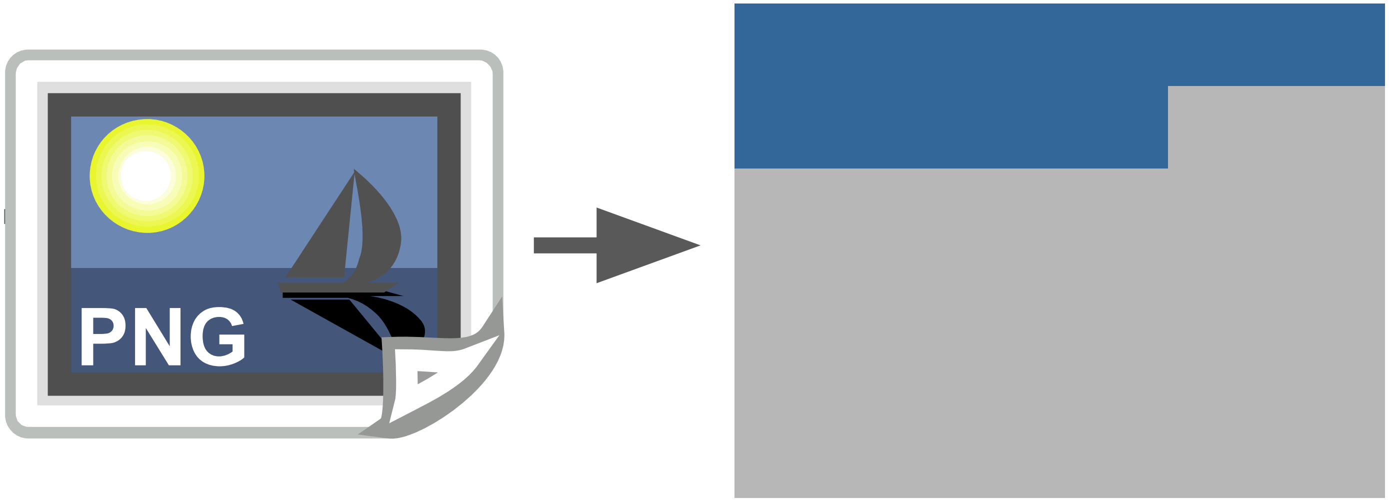
Agenda

- **Why contemporary approaches are insufficient**
  - **Futures**
  - Threads
  - Processes
- Function calls with timeouts
- Backwards compatibility
- Preemptive userland threading

Problem: futures are cooperative

future: lightweight userland thread scheduled by the language runtime

One future can depend on another's result at a *yield point*



Agenda

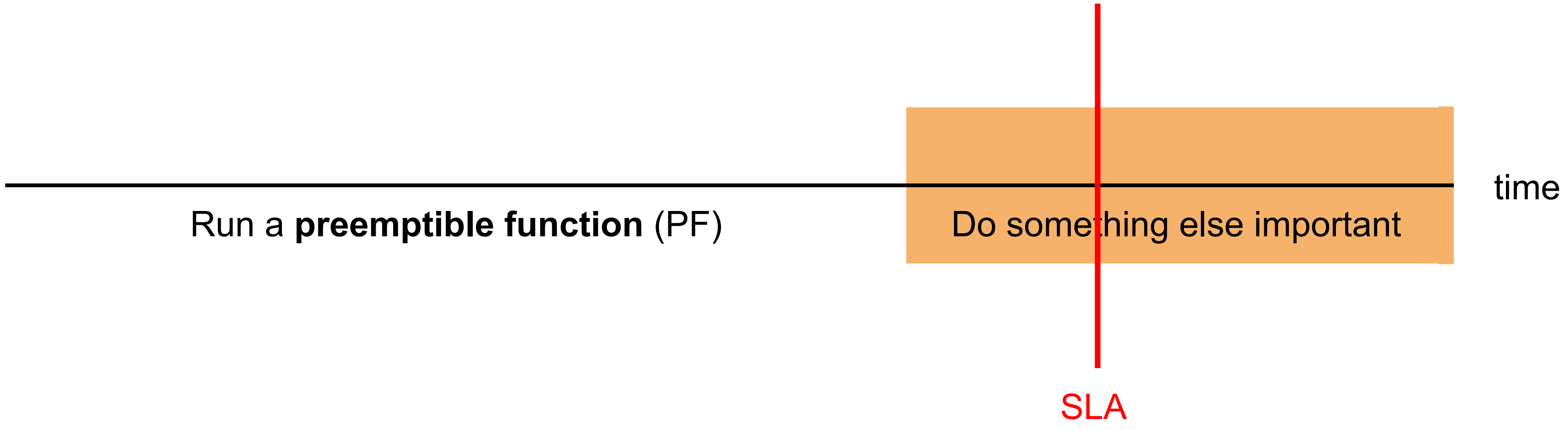
- **Why contemporary approaches are insufficient**
  - ~~Futures~~ (cooperative not preemptive)
  - **Threads**
  - Processes
- Function calls with timeouts
- Backwards compatibility
- Preemptive userland threading

Alternative: kernel threading

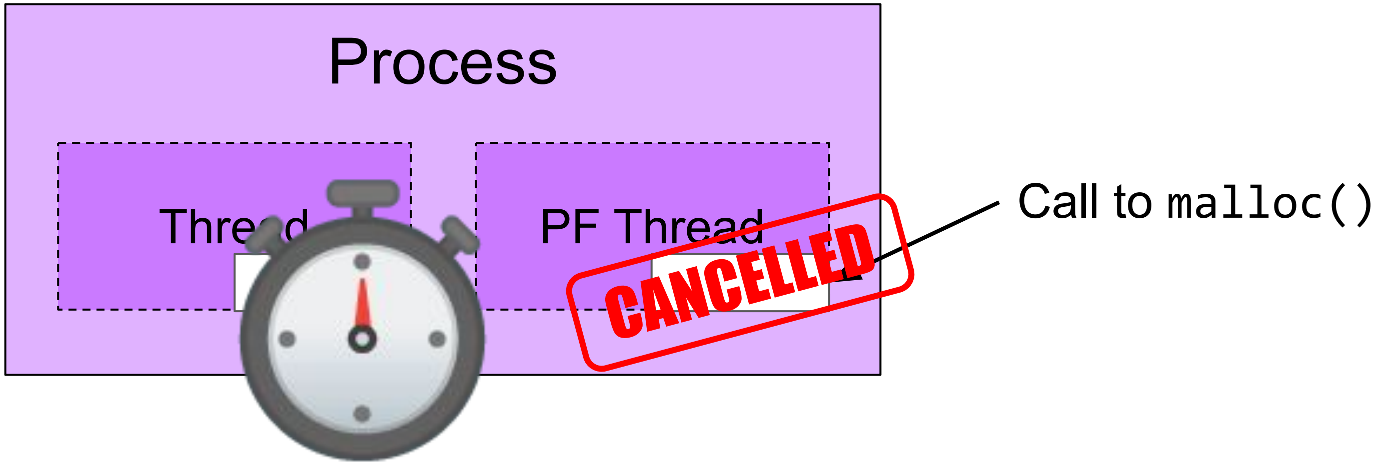
```c
// Problem
buffer = decode(&img);
time_sensitive_task();
```

```c
// Tempting approach
pthread_create(&tid, NULL,
               decode, &img);
usleep(TIMEOUT);
time_sensitive_task();
pthread_join(&tid, &buffer);
```

Problem: SLAs and graceful degradation



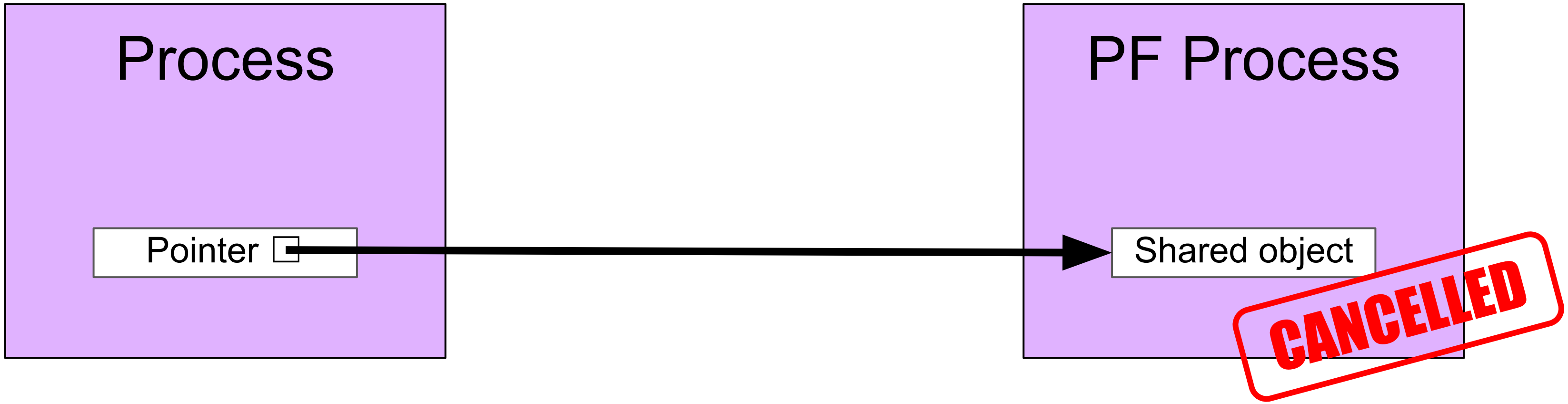
Observation: cancellation is hard



Agenda

- **Why contemporary approaches are insufficient**
  - ~~Futures~~ (cooperative not preemptive)
  - ~~Threads~~ (poor ergonomics, no cancellation)
  - **Processes**
- Function calls with timeouts
- Backwards compatibility
- Preemptive userland threading

Problem: object ownership and lifetime



Agenda

- **Why contemporary approaches are insufficient**
  - ~~Futures~~ (cooperative not preemptive)
  - ~~Threads~~ (poor ergonomics, no cancellation)
  - ~~Processes~~ (poor performance and ergonomics)

  } (sacrifice programmer control)
- Function calls with timeouts
- Backwards compatibility
- Preemptive userland threading

Idea: function calls with timeouts

- Retain programmer's control over the CPU
- Be able to interrupt arbitrary unmodified code
- Introduce minimal overhead in the common case
- Support cancellation
- Maintain compatibility with the existing systems stack

Agenda

- Why contemporary approaches are insufficient
- **Function calls with timeouts**
- Backwards compatibility
- Preemptive userland threading

A new application primitive

lightweight preemptible function: function invoked with a timeout

- Faster than spawning a process or thread
- Runs on the caller’s thread

A new application primitive

lightweight preemptible function: function invoked with a timeout

- Interrupts at 10–100s microseconds granularity
- Pauses on timeout for low overhead and flexibility to resume

A new application primitive

lightweight preemptible function: function invoked with a timeout

- Preemptible code is a normal function or closure
- Invoked via wrapper like `pthread_create()`, but synchronous

The interface: **launch()** and **resume()**

```
funcstate = launch(func, 400 /*us*/, NULL);
if(!funcstate.is_complete) {
    work_queue.push(funcstate);
}

// ...

funcstate = work_queue.pop();
resume(&funcstate, 200 /*us*/);
```

The interface: **cancel()**

```
funcstate = launch(func, 400 /*us*/, NULL);

if(!funcstate.is_complete) {
    work_queue.push(funcstate);
}

// ...

funcstate = work_queue.pop();
cancel(&funcstate);
```

Concurrency: explicit sharing

```
counter = 0;
funcstate = launch(λa. ++counter, 1, NULL);

++counter;

if(!funcstate.is_complete) {
   resume(&funcstate, TO_COMPLETION);
}

assert(counter == 2); // counter == ?!
```

Concurrency: existing protections work (e.g., Rust)

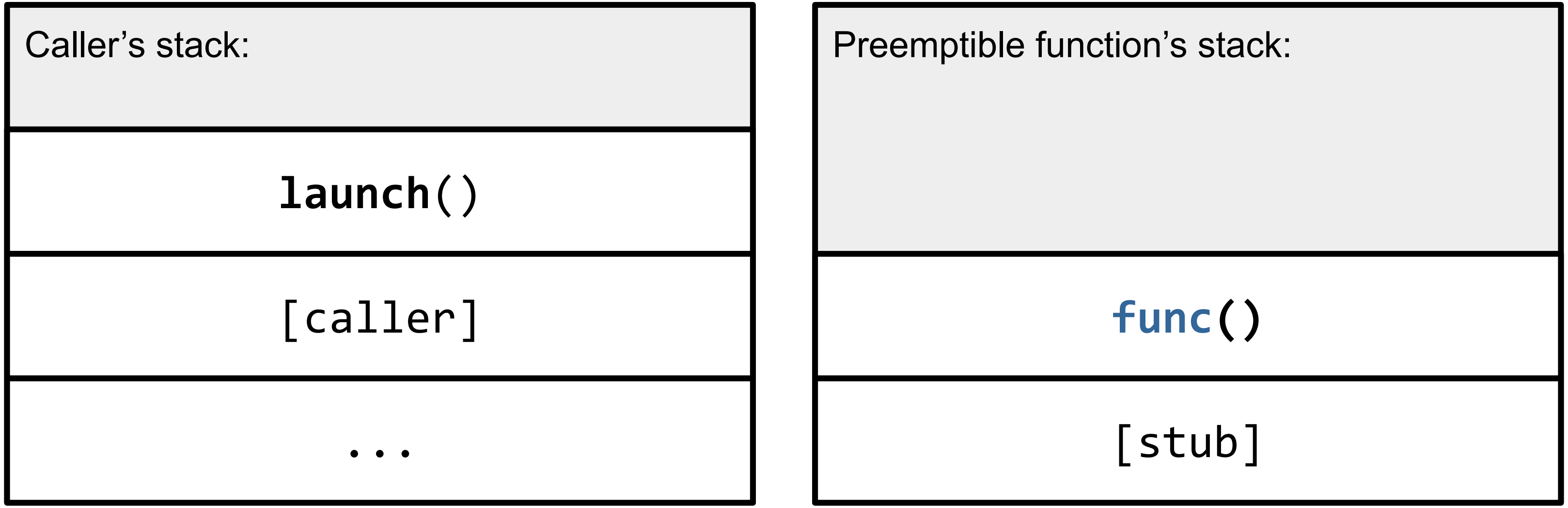
```
error[E0503]: cannot use `counter` because it was mutably borrowed

13 |    funcstate = launch(λa. ++counter, 1, NULL);
   |                       ---   -------- borrow occurs due to use
   |                        |             of `counter` in closure
   |                        |
   |                        borrow of `counter` occurs here
14 |    ++counter;
   |    ^^^^^^^^^ use of borrowed `counter`
```

libinger: library implementing LPFs, currently supports C and Rust programs

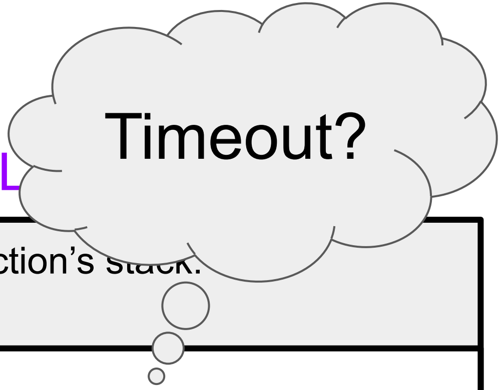
Implementation: execution stack

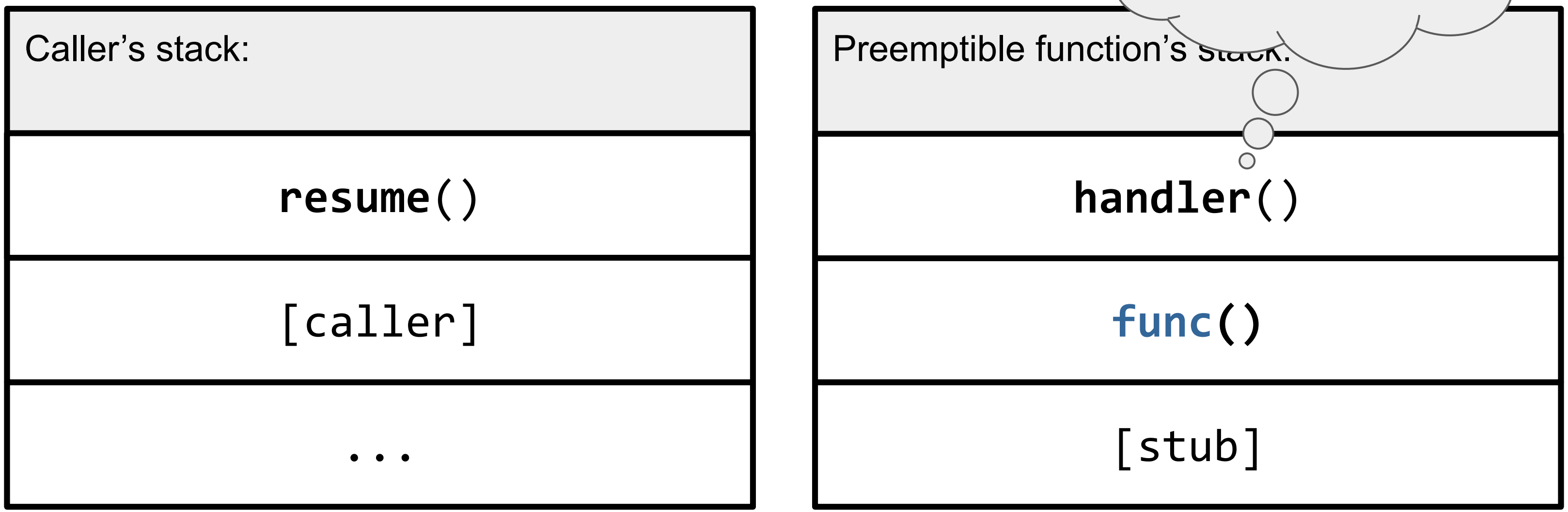
```
funcstate = launch(func, TO_COMPLETION, NULL);
```



Implementation: timer signal

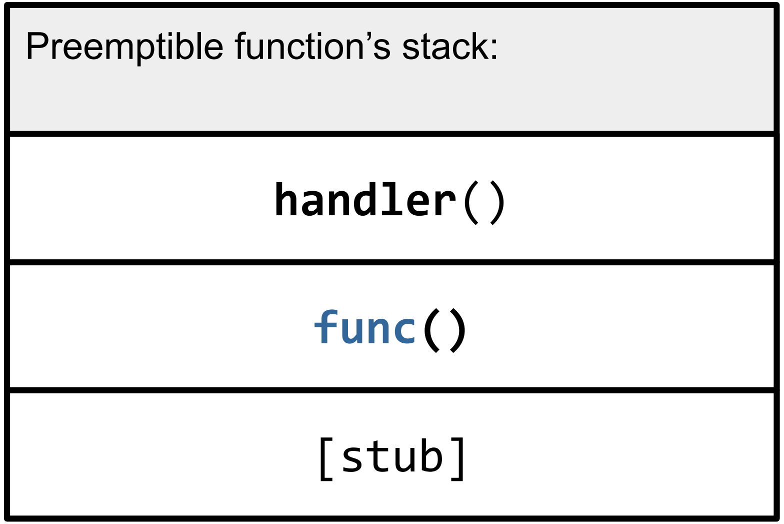
```
funcstate = launch(func, TIMEOUT, NULL
```



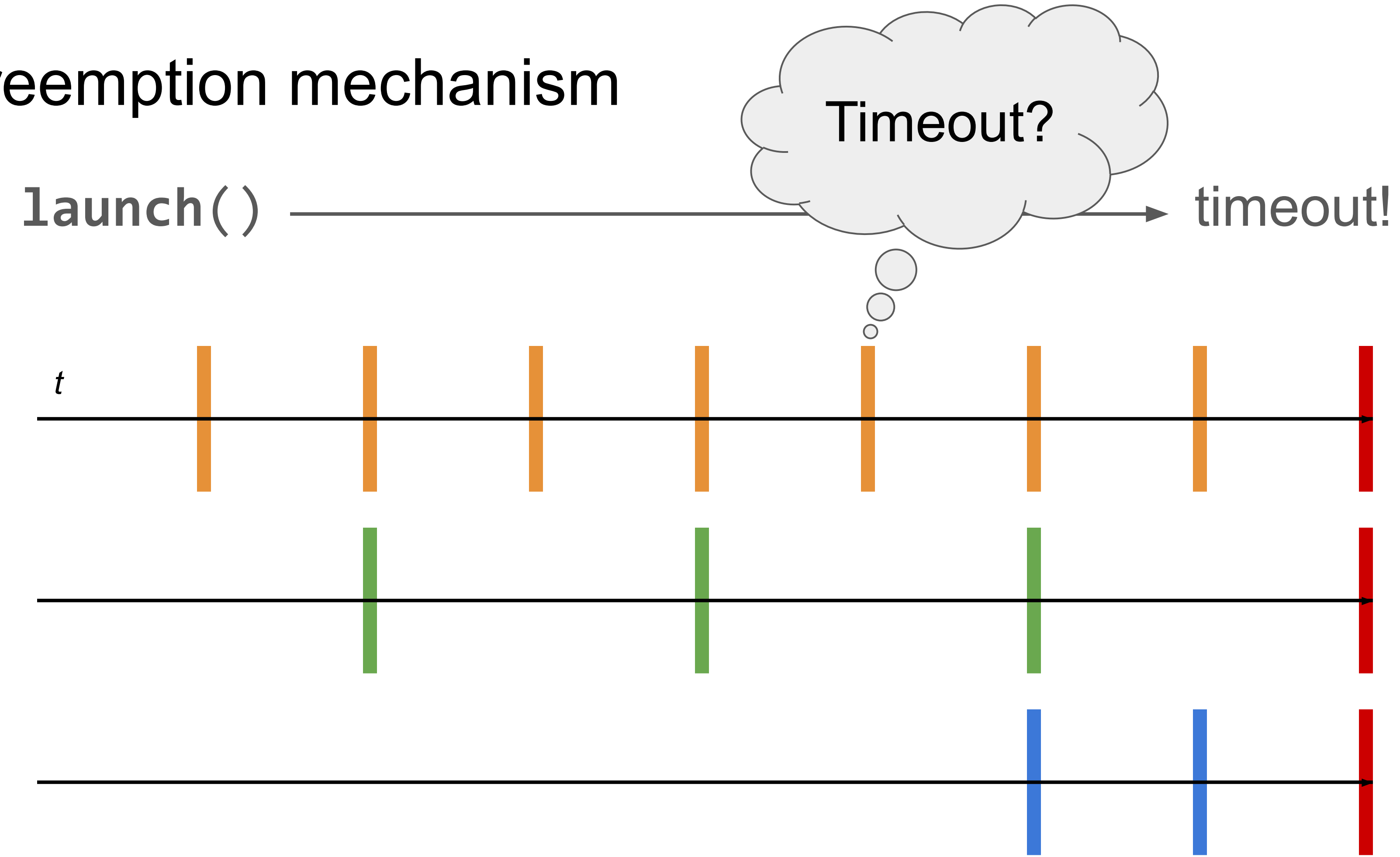


Implementation: cleanup

```
funcstate = launch(func, TIMEOUT, NULL);
cancel(&funcstate);
```



Preemption mechanism

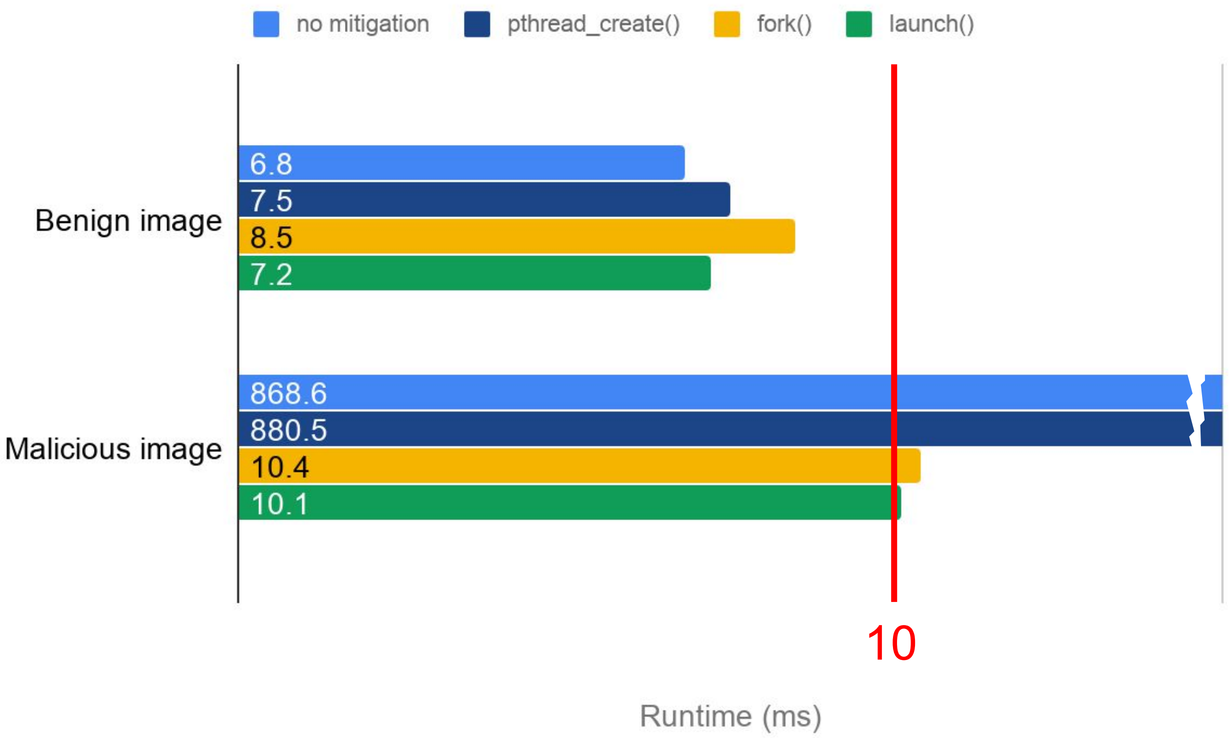


libinger microbenchmarks

Operation	Cost (μs)
`launch()`	≈ 5
`resume()`	≈ 5
`cancel()`	≈ 4800*
`pthread_create()`	≈ 30
`fork()`	≈ 200

* This operation is not typically on the critical path.

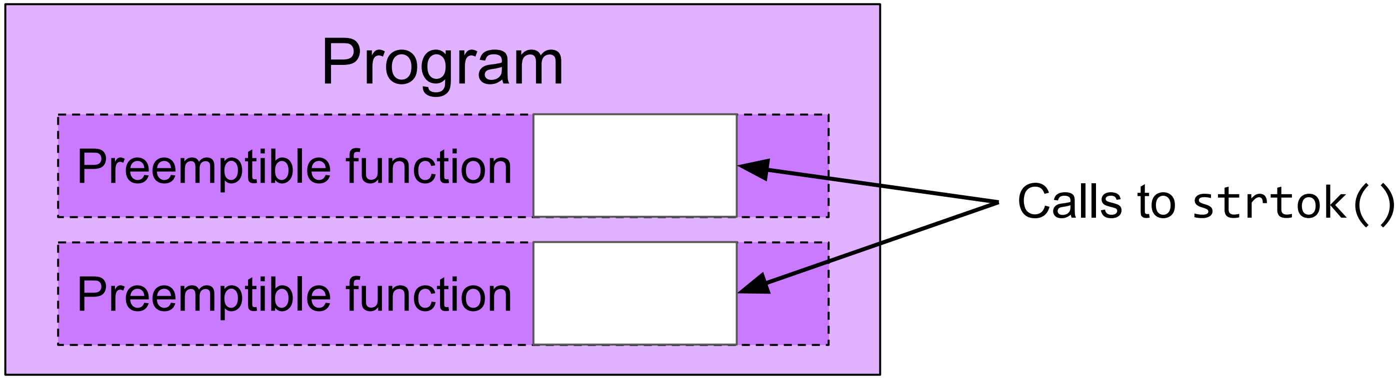
libinger cancels runaway image decoding quickly



Agenda

- Why contemporary approaches are insufficient
- Function calls with timeouts
- **Backwards compatibility**
- Preemptive userland threading

Problem: non-reentrancy

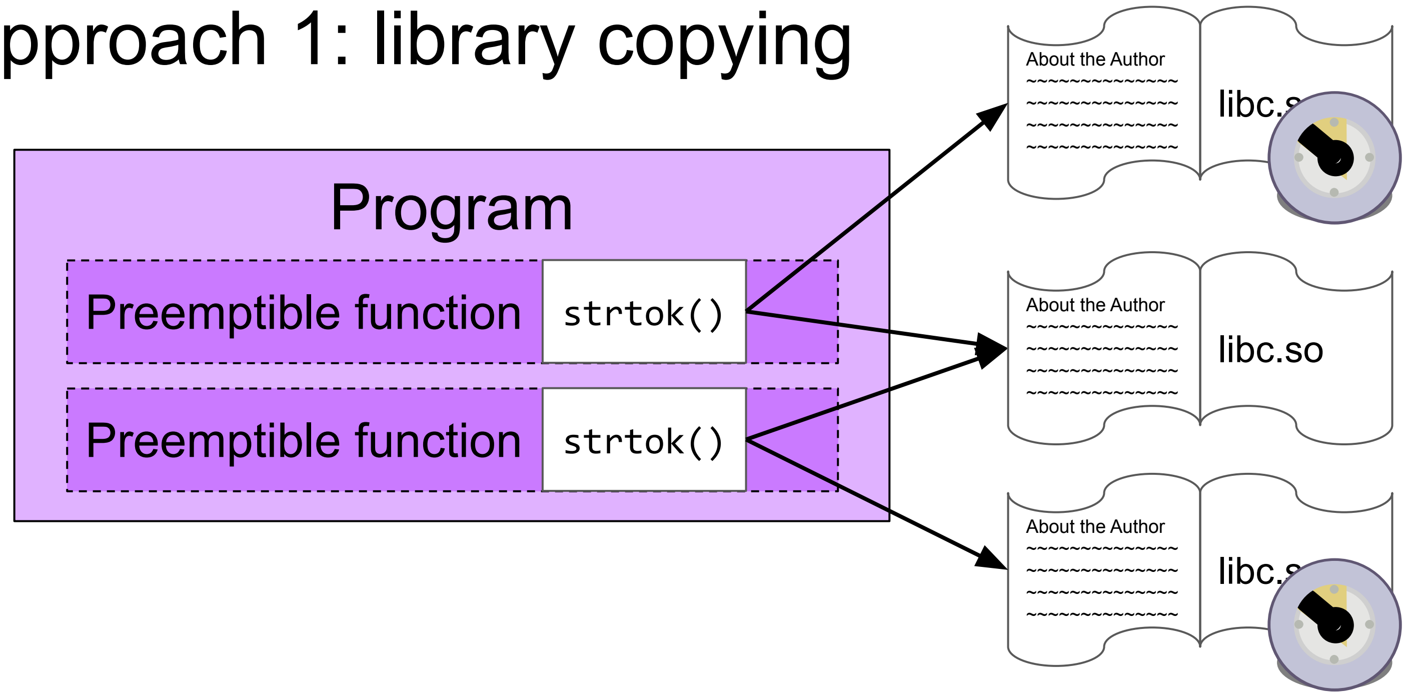


Signal handlers cannot call non-reentrant code

The rest of the program interrupts a preemptible function

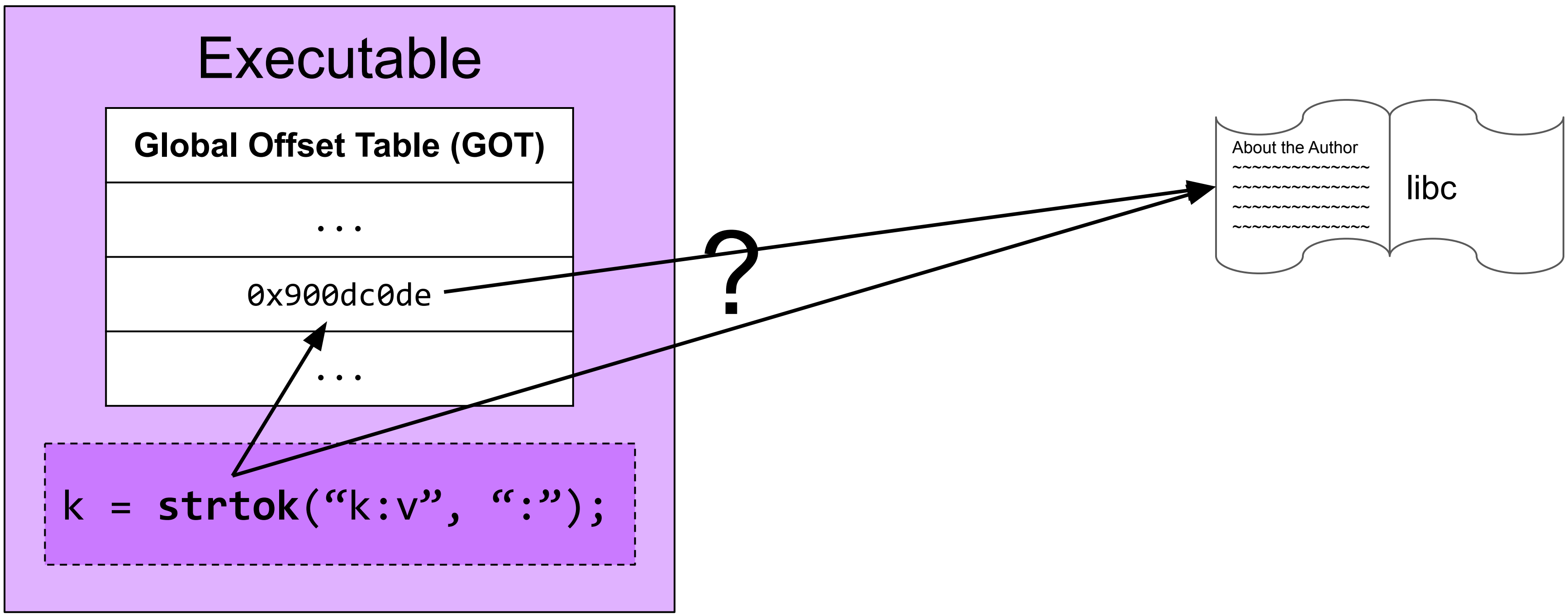
The *rest of the program* cannot call non-reentrant code?!

Approach 1: library copying



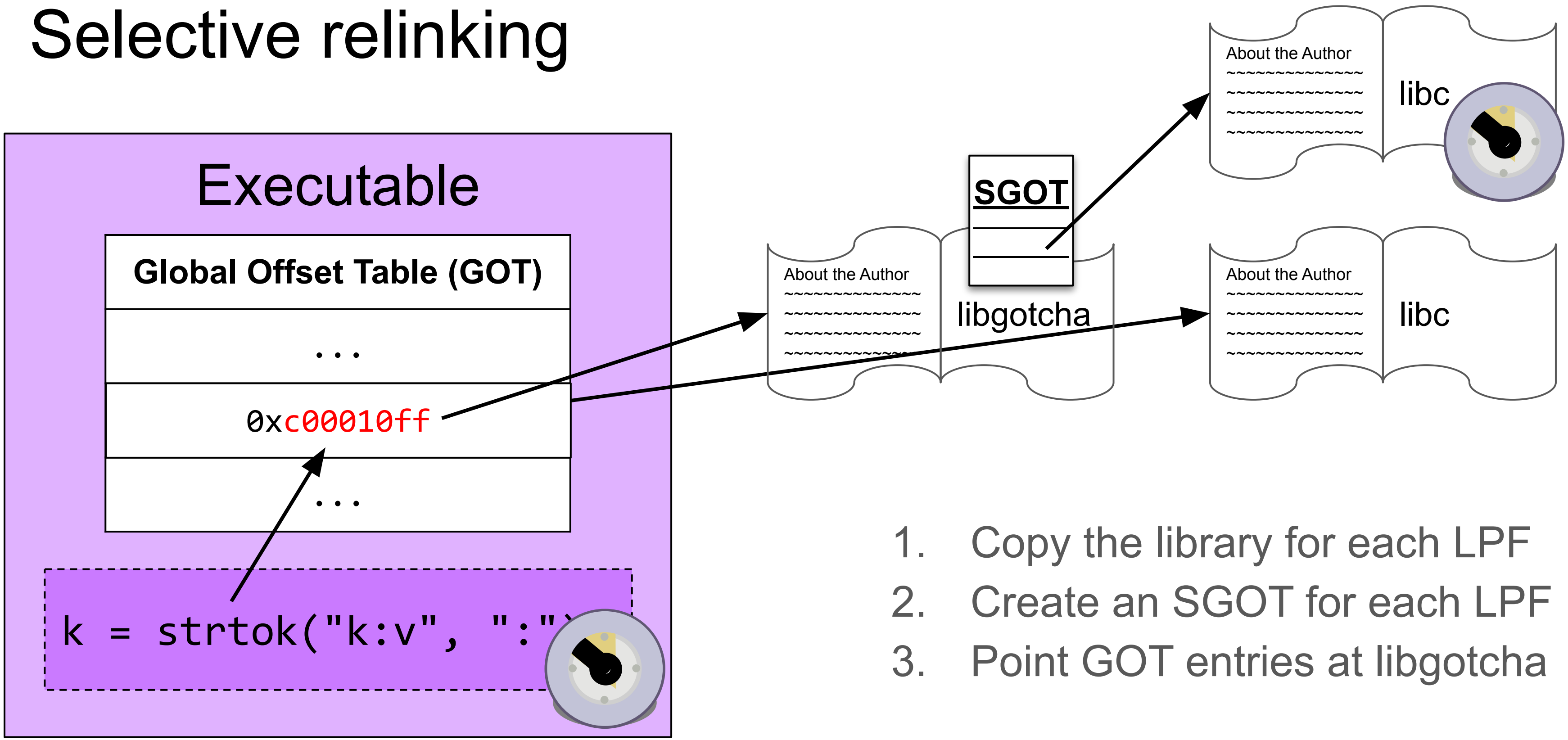
Can reuse each library copy once function runs to completion

Dynamic symbol binding



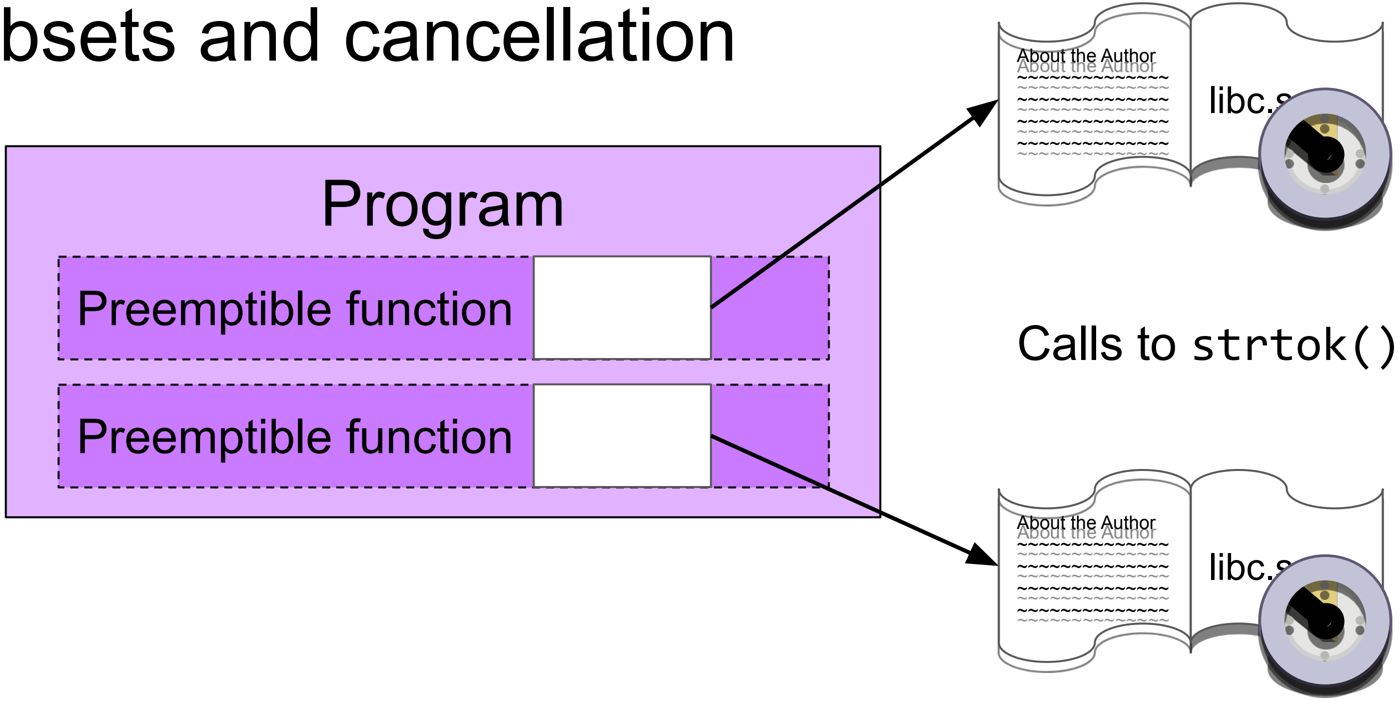
*lib**got**cha:* runtime implementing selective relinking for linked programs

Selective relinking



1. Copy the library for each LPF
2. Create an SGOT for each LPF
3. Point GOT entries at libgotcha

Libsets and cancellation



libset: full set of all a program's libraries

Approach 2: uncopyable functions

Copying doesn't work for everything…

```
void *malloc(size_t size) {
    PREEMPTION_ENABLED = false;
    void *mem = /* Call the real malloc(). */;
    check_for_timeout();
    PREEMPTION_ENABLED = true;
    return mem;
}
```

“Approach 3”: blocking syscalls

```
int open(const char *filename) {
   while(errno == EAGAIN)
      syscall(SYS_open, filename);
}

struct sigaction sa = {};
sa.sa_flags = SA_RESTART;
```

libgotcha microbenchmarks

Symbol access	Time w/o *libgotcha*	Time w/ *libgotcha*
Function call	≈ 2 ns	≈ 14 ns
Global variable	≈ 0 ns	≈ 3500* ns

Baseline	End-to-end time w/o *libgotcha*
`gettimeofday()`	≈ 19 ns (65% overhead)
`getpid()`	≈ 44 ns (30% overhead)

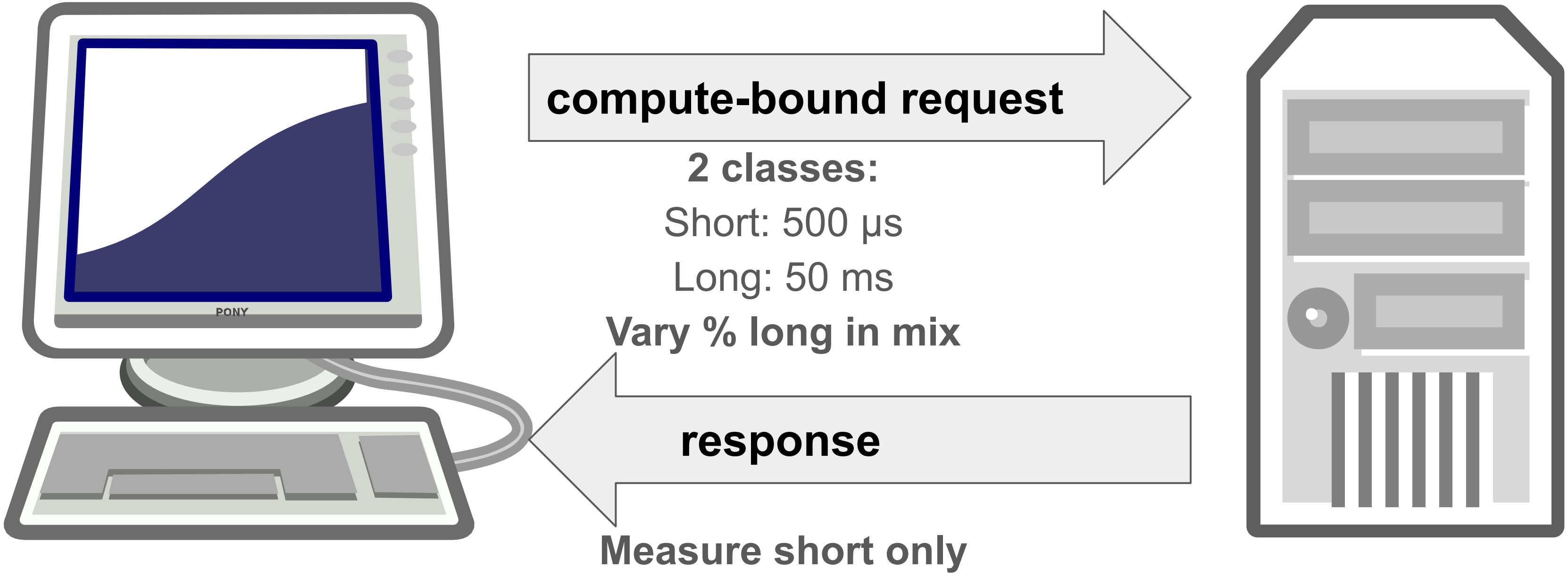
* Exported global variables have become rare.

Agenda

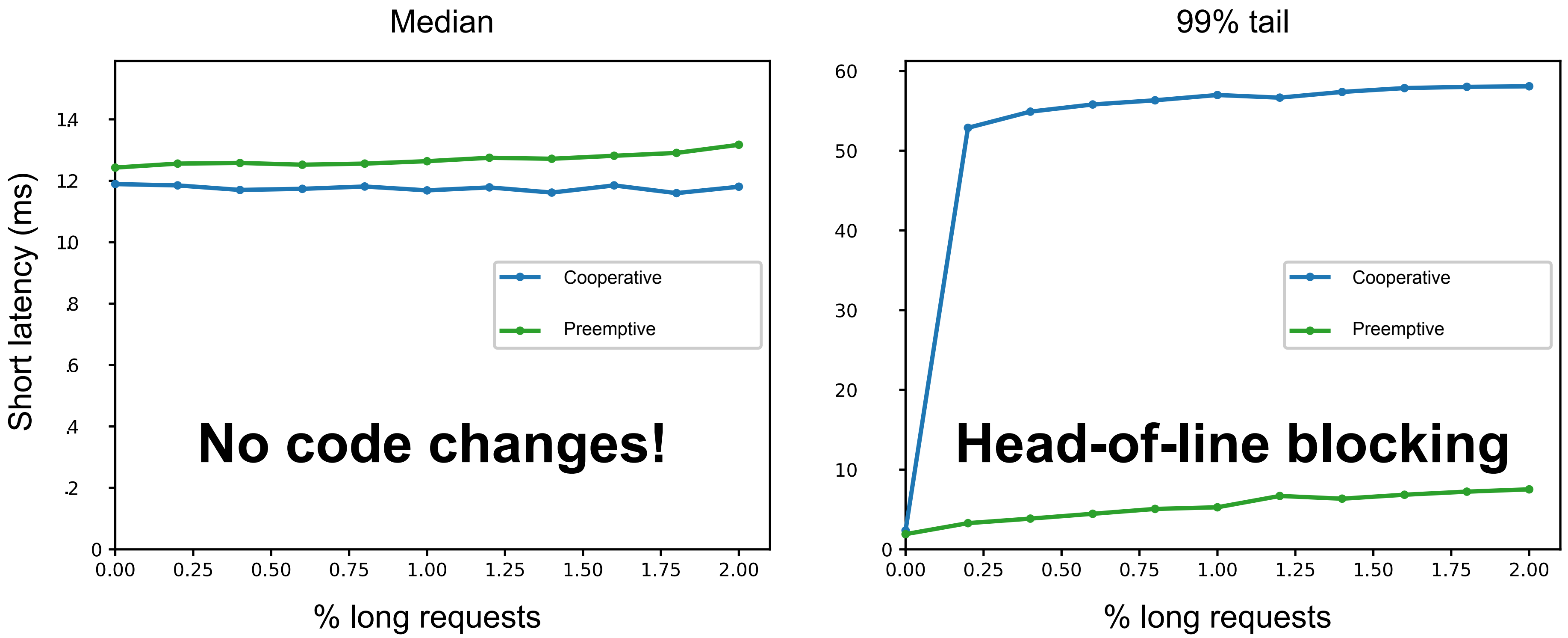
- Why contemporary approaches are insufficient
- Function calls with timeouts
- Backwards compatibility
- **Preemptive userland threading**

libturquoise: preemptive version of the Rust *Tokio* userland thread pool

hyper latency benchmark: experimental setup



hyper latency benchmarks: results



Summary

lightweight preemptible function: function invoked with a timeout

- Synchronous preemption abstraction
- Supports resuming and cancellation
- Interoperable with legacy software
- Exciting systems applications

Thank you!

Reach me at sboucher@cmu.edu