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

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

// Tempting approach
pthread_create(&tid, NULL,
               decode, &img);
usleep(TIMEOUT);
time_sensitive_task();
pthread_join(&tid, &buffer);
Problem: SLAs and graceful degradation

Run a **preemptible function** (PF)

Do something else important

---

SLA

---

time
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

Process

Pointer

PF Process

Shared object

CANCELLED
Agenda

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

- 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()`

```c
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()`

```c
funcstate = launch(func, 400 /*us*/, NULL);
if (!funcstate.is_complete) {
    work_queue.push(funcstate);
}
// ...
funcstate = work_queue.pop();
cancel(&funcstate);
```
Concurrent: explicit sharing

counter = 0;
funcstate = launch(\lambda a. ++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

```rust
13 |   funcstate = launch(\a. ++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

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

<table>
<thead>
<tr>
<th>Caller's stack:</th>
<th>Preemptible function's stack:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>launch()</code></td>
<td></td>
</tr>
<tr>
<td><code>[caller]</code></td>
<td><code>func()</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>[stub]</code></td>
</tr>
</tbody>
</table>
Implementation: timer signal

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

<table>
<thead>
<tr>
<th>Caller's stack:</th>
<th>Preemptible function's stack:</th>
</tr>
</thead>
<tbody>
<tr>
<td>resume()</td>
<td>handler()</td>
</tr>
<tr>
<td>[caller]</td>
<td>func()</td>
</tr>
<tr>
<td></td>
<td>[stub]</td>
</tr>
</tbody>
</table>

Timeout?
Implementation: cleanup

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

Preemptible function’s stack:

handler()

func()

[stub]
Preemption mechanism

launch() → Timeout?

Timeout!

$t$

(timeout)
### libinger microbenchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>launch()</td>
<td>≈ 5</td>
</tr>
<tr>
<td>resume()</td>
<td>≈ 5</td>
</tr>
<tr>
<td>cancel()</td>
<td>≈ 4800*</td>
</tr>
<tr>
<td>pthread_create()</td>
<td>≈ 30</td>
</tr>
<tr>
<td>fork()</td>
<td>≈ 200</td>
</tr>
</tbody>
</table>

* 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

Program

Preemptible function `strtok()`

Preemptible function `strtok()`

Can reuse each library copy once function runs to completion
Dynamic symbol binding

Executable

<table>
<thead>
<tr>
<th>Global Offset Table (GOT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>0x900dc0de</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

\[k = \text{strtok("k:v", ";")};\]
libgotcha: runtime implementing selective relinking for linked programs
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...

```c
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

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

struct sigaction sa = {};  
sa.sa_flags = SA_RESTART;
```
**libgotcha** microbenchmarks

<table>
<thead>
<tr>
<th>Symbol access</th>
<th>Time w/o <strong>libgotcha</strong></th>
<th>Time w/ <strong>libgotcha</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Function call</td>
<td>≈ 2 ns</td>
<td>≈ 14 ns</td>
</tr>
<tr>
<td>Global variable</td>
<td>≈ 0 ns</td>
<td>≈ 3500* ns</td>
</tr>
</tbody>
</table>

**Baseline**

<table>
<thead>
<tr>
<th></th>
<th>End-to-end time w/o <strong>libgotcha</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>gettimeofday()</td>
<td>≈ 19 ns (65% overhead)</td>
</tr>
<tr>
<td>getpid()</td>
<td>≈ 44 ns (30% overhead)</td>
</tr>
</tbody>
</table>

* 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

2 classes:
- Short: 500 μs
- Long: 50 ms

Vary % long in mix

Measure short only

compute-bound request

response
**hyper** latency benchmarks: results

**No code changes!**

**Head-of-line blocking**
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