The benefits and costs of writing a POSIX kernel in a high-level language

Cody Cutler, M. Frans Kaashoek, Robert T. Morris

MIT CSAIL

Should we use high-level languages to build OS kernels?

HLL Benefits

- Easier to program
- Simpler concurrency with GC
- Prevents classes of kernel bugs

Kernel memory safety matters

Inspected Linux kernel execute code CVEs for 2017

40 CVEs due to just memory-safety bugs

Kernel memory safety matters

Inspected Linux kernel execute code CVEs for 2017

40 CVEs due to just memory-safety bugs

HLL would have prevented code execution

HLL downside: safety costs performance

- Bounds, cast, nil-pointer checks
- Reflection
- Garbage collection

Goal: measure HLL impact

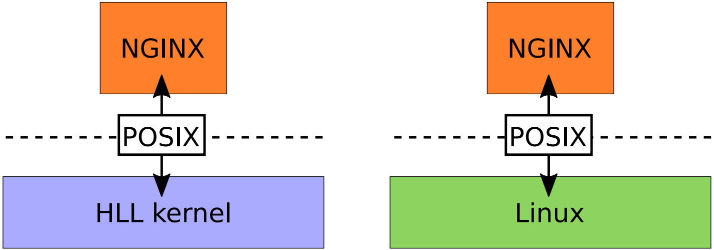
Pros:

- Reduction of bugs
- Simpler code

Cons:

- HLL safety tax
- GC CPU and memory overhead
- GC pause times

Methodology



Build new HLL kernel, compare with Linux

Isolate HLL impact:

Same apps, POSIX interface, and monolithic organization

Previous work

Taos*(ASPLOS'87)*, Spin*(SOSP'95)*, Singularity*(SOSP'07)*, Tock*(SOSP'17)*, J-kernel*(ATC'98)*, KaffeOS*(ATC'00)*, House*(ICFP'05)*,...

- Explore new ideas
- Different architectures

Several studies of HLL versus C for user programs

- Kernels different from user programs

Previous work

Taos*(ASPLOS'87)*, Spin*(SOSP'95)*, Singularity*(SOSP'07)*, Tock*(SOSP'17)*, J-kernel*(ATC'98)*, KaffeOS*(ATC'00)*, House*(ICFP'05)*,...

- Explore new ideas
- Different architectures

Several studies of HLL versus C for user programs

- Kernels different from user programs

None measure HLL impact in a monolithic POSIX kernel

Contributions

BISCUIT, new x86-64 Go kernel

- Runs unmodified Linux applications
- with good performance

Measurements of HLL costs for NGINX, Redis, and CMailbench

Description of qualitative ways HLL helped

New scheme to deal with heap exhaustion

Which HLL?

Go is a good choice:

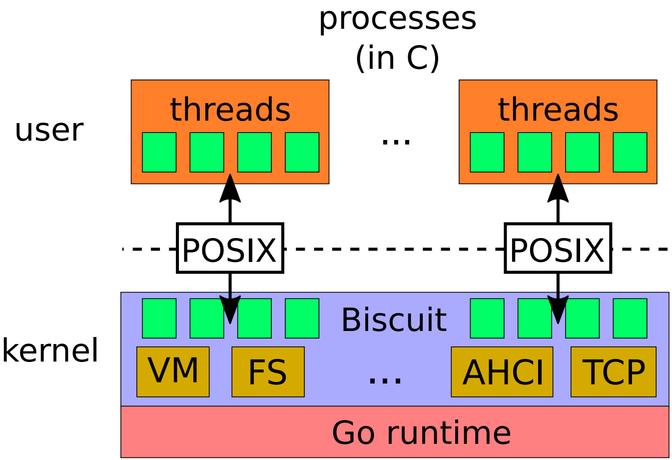
- Easy to call asm
- Compiled to machine code w/good compiler
- Easy concurrency
- Easy static analysis
- GC

Go's GC

Concurrent mark and sweep

Stop-the-world pauses of 10s of μs

BISCUIT overview



58 syscalls, LOC: *28k Go,*
1.5k assembly (boot, entry/exit)

Features

- Multicore
- Threads
- Journaled FS (7k LOC)
- Virtual memory (2k LOC)
- TCP/IP stack (5k LOC)
- Drivers: AHCI and Intel 10G NIC (3k LOC)

No fundamental challenges due to HLL

But many implementation puzzles

- Interrupts
- Kernel threads are lightweight
- Runtime on bare-metal
- ...

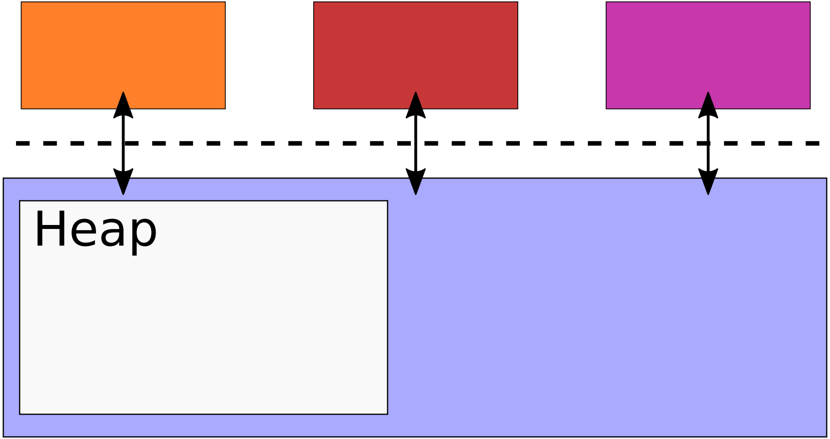
No fundamental challenges due to HLL

But many implementation puzzles

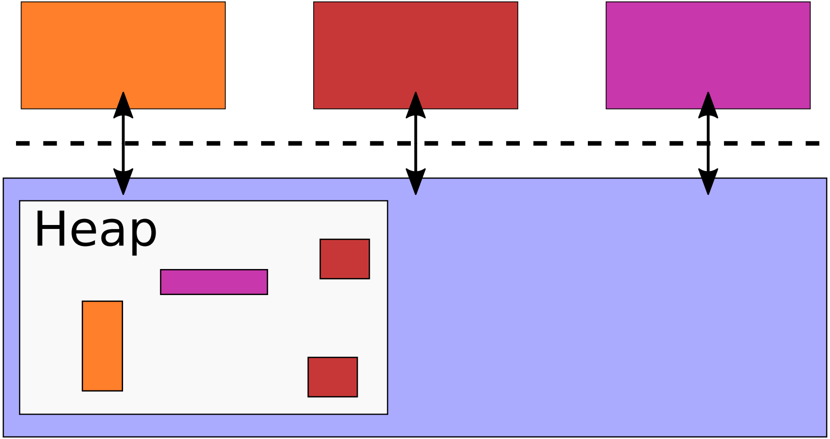
- Interrupts
- Kernel threads are lightweight
- Runtime on bare-metal
- ...

Surprising puzzle: heap exhaustion

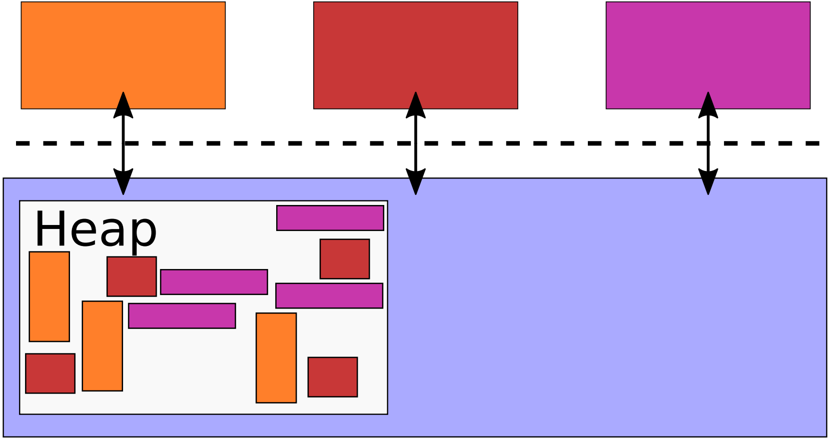
Puzzle: Heap exhaustion



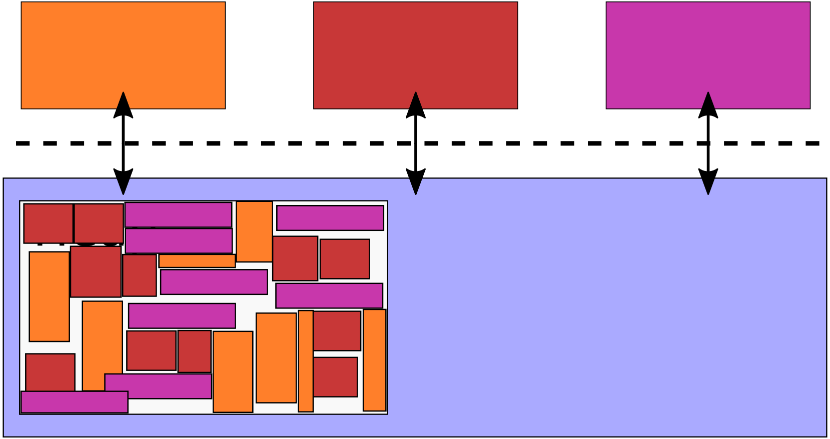
Puzzle: Heap exhaustion



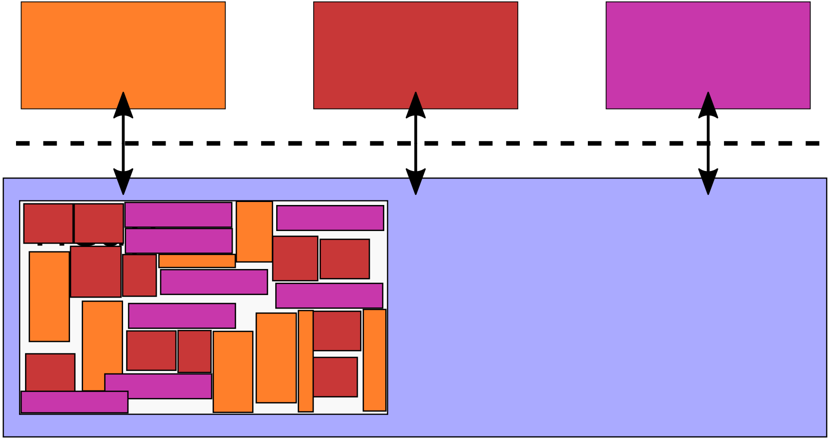
Puzzle: Heap exhaustion



Puzzle: Heap exhaustion



Puzzle: Heap exhaustion



Can't allocate heap memory $\Longrightarrow$ nothing works

All kernels face this problem

How to recover?

Strawman 1: Wait for memory in allocator?

How to recover?

Strawman 1: Wait for memory in allocator?

- May deadlock!

How to recover?

Strawman 1: Wait for memory in allocator?

- May deadlock!

Strawman 2: Check/handle allocation failure, like C kernels?

How to recover?

Strawman 1: Wait for memory in allocator?

- May deadlock!

Strawman 2: Check/handle allocation failure, like C kernels?

- Difficult to get right

How to recover?

Strawman 1: Wait for memory in allocator?

- May deadlock!

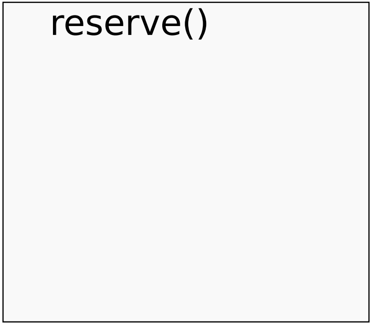
Strawman 2: Check/handle allocation failure, like C kernels?

- Difficult to get right
- Can't! Go doesn't expose failed allocations
- and implicitly allocates

Both cause problems for Linux; see "too small to fail" rule

BISCUIT solution: reserve memory

To execute syscall...



BISCUIT solution: reserve memory

To execute syscall...

reserve()

(no locks held)

BISCUIT solution: reserve memory

To execute syscall...

```
reserve()
    (no locks held)
    evict, kill
    wait...
```

BISCUIT solution: reserve memory

To execute syscall...

```
reserve()
    (no locks held)
    evict, kill
    wait...
sys_read()
    ...
```

BISCUIT solution: reserve memory

To execute syscall...

```
reserve()
    (no locks held)
    evict, kill
    wait...
sys_read()
    ...
unreserve()
```

BISCUIT solution: reserve memory

To execute syscall...

```
reserve()
    (no locks held)
    evict, kill
    wait...
sys_read()
    ...
unreserve()
```

No checks, no error handling code, no deadlock

Reservations

HLL easy to analyze

Tool computes reservation via escape analysis

- Using Go's static analysis packages

$\approx$ three days of expert effort to apply tool

Building BISCUIT was similar to other kernels

Building BISCUIT was similar to other kernels

BISCUIT adopted many Linux optimizations:

- large pages for kernel text
- per-CPU NIC transmit queues
- RCU-like directory cache
- concurrent FS transactions
- pad structs to remove false sharing

Good OS performance more about optimizations, less about HLL

Eval questions

Should we use high-level languages to build OS kernels?

1. Did BISCUIT benefit from HLL features?
2. Is BISCUIT performance in the same league as Linux?
3. What is the breakdown of HLL tax?
4. What is the performance cost of Go compared to C?

More experiments in paper

1: Qualitative benefits of HLL features

Simpler code with:

- GC'ed allocation
- `defer`
- multi-valued return
- closures
- maps

HLL example benefits

Example 1: Memory safety

Example 2: Simpler concurrency

1: BISCUIT benefits from memory safety

Inspected fixes for all publicly-available execute code CVEs in Linux kernel for 2017

Category	#	Outcome in Go
—	11	unknown
logic	14	same
use-after-free/double-free	8	disappear due to GC
out-of-bounds	32	`panic` or disappear due to GC

`panic` likely better than malicious code execution

1: BISCUIT benefits from simpler concurrency

Generally, concurrency with GC simpler

Particularly, GC greatly simplifies read-lock-free data structures

Challenge: In C, how to determine when last reader is done?

Main purpose of read-copy update (RCU) (*PDCS'98*)
Linux uses RCU, but it's not easy

- Code to start and end RCU sections
- No sleeping/scheduling in RCU sections
- ...

In Go, no extra code — GC takes care of it

Experimental setup

Hardware:

- 4 core 2.8Ghz Xeon-X3460
- 16 GB RAM
- Hyperthreads disabled

Eval application:

- NGINX (1.11.5) – webserver
- Redis (3.0.5) – key/value store
- CMailbench – mail-server benchmark

Applications are kernel intensive

No idle time

79%-92% kernel time

In-memory FS

Run for a minute

512MB heap RAM for BISCUIT

2: Is BISCUIT perf in the same league as Linux?

Debian 9.4, Linux 4.9.82

Disabled expensive features:

- page-table isolation
- retpoline
- kernel address space layout randomization
- transparent huge-pages
- ...

2: Biscuit is in the same league

	Biscuit ops/s	**Linux ops/s**	**Ratio**
CMailbench (mem)	15,862	17,034	1.07
NGINX	88,592	94,492	1.07
Redis	711,792	775,317	1.09

2: Biscuit is in the same league

	Biscuit ops/s	Linux ops/s	Ratio
CMailbench (mem)	15,862	17,034	1.07
NGINX	88,592	94,492	1.07
Redis	711,792	775,317	1.09

HLL cost unclear from comparison

May understate Linux performance due to features:

- NUMA awareness
- Optimizations for large number of cores (>4)
- ...

Focus on HLL costs:

- Measure CPU cycles BISCUIT pays for HLL tax
- Compare code paths that differ only by language

3: What is the breakdown of HLL tax?

Measure HLL tax:

- GC cycles
- Prologue cycles
- Write barrier cycles
- Safety cycles

3: Prologue cycles are most expensive

	GC cycles	GCs	Prologue cycles	Write barrier cycles	Safety cycles
CMailbench	3%	42	6%	$< 1\%$	3%
NGINX	2%	32	6%	$< 1\%$	2%
Redis	1%	30	4%	$< 1\%$	2%

3: Prologue cycles are most expensive

	GC cycles	GCs	Prologue cycles	Write barrier cycles	Safety cycles
CMailbench	3%	42	6%	$<$ 1%	3%
NGINX	2%	32	6%	$<$ 1%	2%
Redis	1%	30	4%	$<$ 1%	2%

3: Prologue cycles are most expensive

	GC cycles	GCs	Prologue cycles	Write barrier cycles	Safety cycles
CMailbench	3%	42	6%	$< 1\%$	3%
NGINX	2%	32	6%	$< 1\%$	2%
Redis	1%	30	4%	$< 1\%$	2%

3: Prologue cycles are most expensive

	GC cycles	GCs	Prologue cycles	Write barrier cycles	Safety cycles
CMailbench	3%	42	6%	$< 1\%$	3%
NGINX	2%	32	6%	$< 1\%$	2%
Redis	1%	30	4%	$< 1\%$	2%

3: Prologue cycles are most expensive

	GC cycles	GCs	Prologue cycles	Write barrier cycles	Safety cycles
CMailbench	3%	42	6%	$< 1\%$	3%
NGINX	2%	32	6%	$< 1\%$	2%
Redis	1%	30	4%	$< 1\%$	2%

Benchmarks allocate kernel heap rapidly
but have little persistent kernel heap data

Cycles used by GC increase with size of live kernel heap
Dedicate 2 or 3$\times$ memory $\Rightarrow$ low GC cycles

4: What is the cost of Go compared to C?

Make code paths same in BISCUIT and Linux

Two code paths in paper

- pipe ping-pong (systems calls, context switching)
- page-fault handler (exceptions, VM)

Focus on pipe ping-pong:

- LOC: 1.2k Go, 1.8k C
- No allocation; no GC
- Top-10 most expensive instructions match

4: C is 15% faster

C (ops/s)	Go (ops/s)	Ratio
536,193	465,811	1.15

Prologue/safety-checks $\Rightarrow$ 16% more instructions

Should one use HLL for a new kernel?

The HLL worked well for kernel development

Performance is paramount $\Rightarrow$ use C (up to 15%)

Minimize memory use $\Rightarrow$ use C ($\downarrow$ mem. budget, $\uparrow$ GC cost)

Safety is paramount $\Rightarrow$ use HLL (40 CVEs stopped)

Performance merely important $\Rightarrow$ use HLL (pay 15%, memory)

Questions?

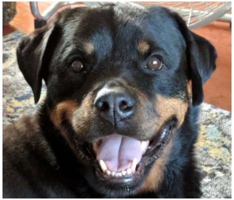
The HLL worked well for kernel development

Performance is paramount $\Rightarrow$ use C (up to 15%)

Minimize memory use $\Rightarrow$ use C ($\downarrow$ mem. budget, $\uparrow$ GC cost)

Safety is paramount $\Rightarrow$ use HLL (40 CVEs stopped)

Performance merely important $\Rightarrow$ use HLL (pay 15%, memory)



git clone https://github.com/mit-pdos/biscuit.git