

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

---

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

MIT CSAIL

## What language to use for developing a kernel?

A hotly-debated question but often with few facts

6.1810/6.828/6.S081 students: why are we using C? why not a type-safe language?

To shed some light, we wrote a *new* kernel with

- A language with automatic memory management (i.e., with a garbage collector)
- A traditional, monolithic Unix organization

## C is popular for kernels

Windows

Linux

\*BSD

## Why C is good: complete control

Control of memory allocation and freeing

Almost no implicit, hidden code

Direct access to memory

Few dependencies

## Why C is bad

Writing secure C code is difficult

- buffer overruns
- use-after-free bugs
- threads sharing dynamic memory

40 Linux kernel execute-code CVEs in 2017 due to memory-safety errors

(execute-code CVE is a bug that enables attacker to run malicious code in kernel)

## High-level languages (HLLs) provide memory-safety

All 40 CVEs would not execute malicious code in an HLL

## HLL benefits

Type safety

Automatic memory management with garbage collector

Concurrency

Abstraction

## HLL potential downsides

Poor performance:

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

Incompatibility with kernel programming:

- No direct memory access
- No hand-written assembly
- Limited concurrency or parallelism



## Goal: measure HLL trade-offs

Explore total effect of using HLL instead of C:

- Impact on safety
- Impact on programmability
- Performance cost

...for production-grade kernel

## Prior work: HLL trade-offs

Many studies of HLL trade-offs for user programs (*Hertz'05, Yang'04*)

But kernels different from user programs

(ex: more careful memory management)

Need to measure HLL trade-offs in kernel

## Prior work: HLL kernels

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

Explore new ideas and architectures

None measure HLL trade-offs vs C kernel

## Measuring trade-offs is tricky

Must compare with production-grade C kernel (e.g., Linux)

Problem: can't build production-grade HLL kernel

## The most we can do

Build HLL kernel

Keep important parts the same as Linux

Optimize until performance is roughly similar to Linux

Measure HLL trade-offs

Risk: measurements of production-grade kernels differ

## Methodology



Built HLL kernel

Same apps, POSIX interface, and monolithic organization

Optimized, measured HLL trade-offs

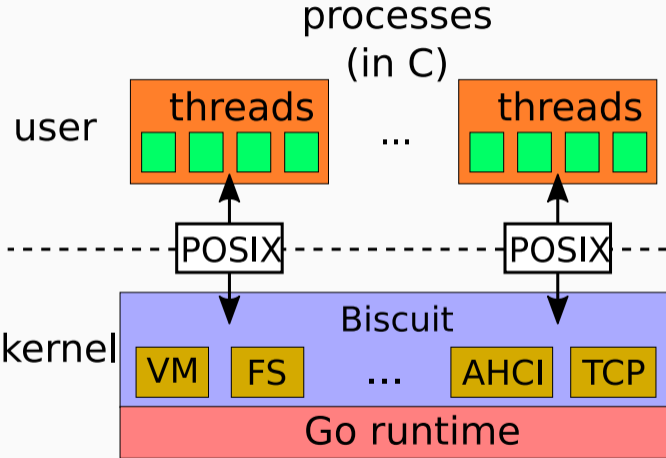
## Which HLL?

Go is a good choice:

- Easy to call assembly
- Compiled to machine code w/good compiler
- Easy concurrency
- Easy static analysis
- GC (Concurrent mark and sweep)

Rust might be a fine choice too

# BISCUIT overview





## BISCUIT Features

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

## User programs

Process has own address space

User/kernel memory isolated by hardware

Each user thread has companion kernel thread

Kernel threads are “goroutines”

## System calls

User thread put args in registers

User thread executes *SYSENTER*

Control passes to kernel thread

Kernel thread executes system call, returns via *SYSEXIT*

## BISCUIT design puzzles

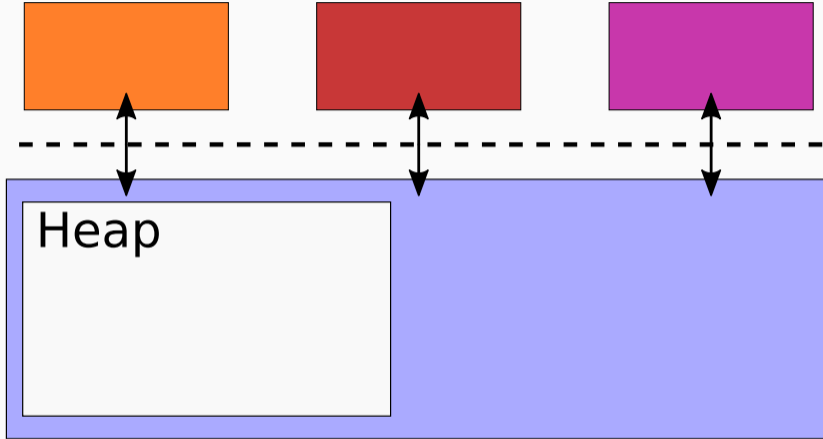
Runtime on bare-metal

Goroutines run different applications

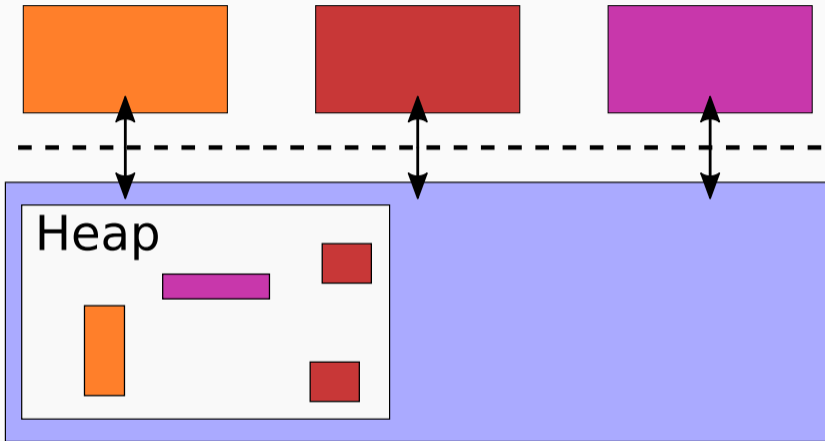
Device interrupts in runtime critical sections

Hardest puzzle: heap exhaustion

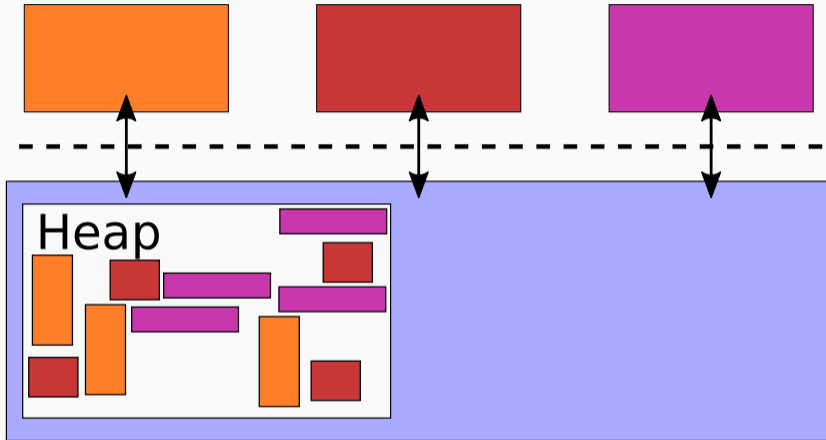
## Puzzle: Heap exhaustion



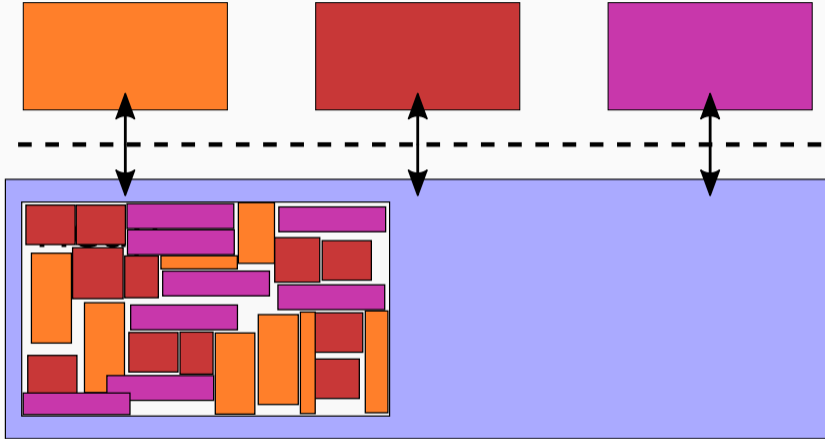
## Puzzle: Heap exhaustion



## Puzzle: Heap exhaustion

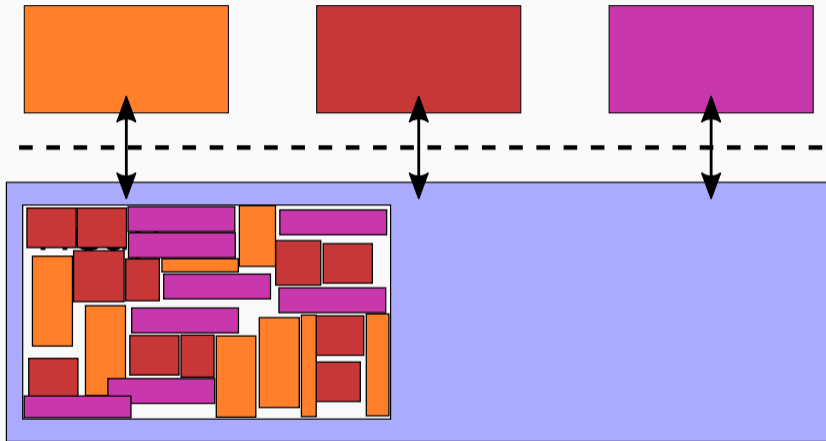


# Puzzle: Heap exhaustion





## Puzzle: Heap exhaustion



Can't allocate heap memory  $\implies$  nothing works  
All kernels face this problem

## How to recover?

Strawman 0: panic (xv6)

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 implicitly allocates
- Doesn't expose failed allocations

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

## BISCUIT solution: reserve memory

To execute system call...

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

No checks, no error handling code, no deadlock

## Heap reservation bounds

How to compute max memory for each system call?

Smaller heap bounds  $\implies$  more concurrent system calls

## Heap bounds via static analysis

HLL easy to analyze

Tool computes reservation via escape analysis

Using Go's static analysis packages

Annotations for difficult cases

≈ three days of expert effort to apply tool

## BISCUIT implementation

Building BISCUIT was similar to other kernels

## BISCUIT implementation

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
- execute FS ops concurrently with commit
- pad structs to remove false sharing

Good OS performance more about optimizations, less about HLL

# Evaluation

Part 1: HLL benefits

Part 2: HLL performance costs



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

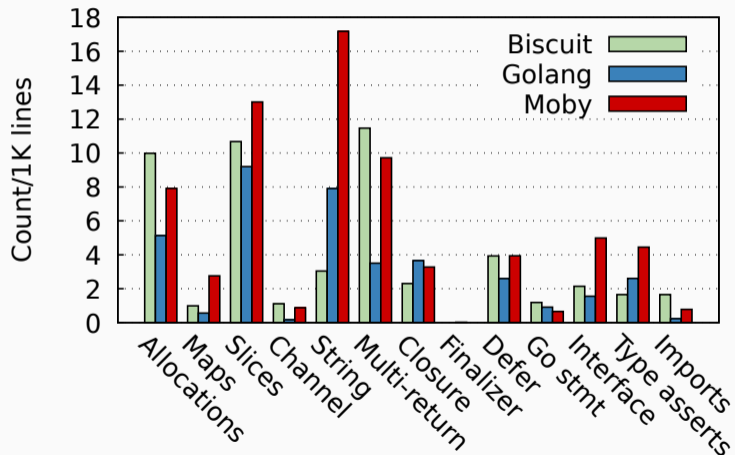
- 1 Does BISCUIT use HLL features?
- 2 Does HLL simplify BISCUIT code?
- 3 Would HLL prevent kernel exploits?

## 1: Does BISCUIT use HLL features?

Counted HLL feature use in BISCUIT and two huge Go projects

(Moby and Golang, >1M LOC)

# 1: BISCUIT uses HLL features



## 2: Does HLL simplify BISCUIT code?

Qualitatively, my favorite features:

- GC'ed allocation
- slices
- defer
- multi-valued return
- strings
- closures
- maps

Net effect: simpler code

## 2: Simpler concurrency

Simpler data sharing between threads

In HLL, GC frees memory

In C, programmer must free memory

## 2: Simpler concurrency example

```
buf := new(object_t)
// Initialize buf...

go func() {
    process1(buf)
}()
process2(buf)
// When should C code free(buf)?
```

## 2: Simpler read-lock-free concurrency

Locks and reference counts expensive in hot paths

Good for performance to avoid them

Challenge in C: when is object free?

## 2: Read-lock-free example

```
var Head *Node
```

```
func get() *Node {  
    return atomic_load(&Head)  
}
```

```
func pop() {  
    Lock()  
    v := Head  
    if v != nil {  
        atomic_store(&Head, v.next)  
    }  
    Unlock()  
}
```



## 2: Simpler read-lock-free concurrency

Linux safely frees via RCU (*McKenney'98*)

Defers free until all CPUs context switch

Programmer must follow RCU rules:

- Prologue and epilogue surrounding accesses
- No sleeping or scheduling

Error prone in more complex situations

GC makes these challenges disappear

HLL significantly simplifies read-lock-free code

### 3: Would HLL prevent kernel exploits?

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

Classify based on outcome of bug in BISCUIT

### 3: HLL prevents kernel exploits

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

panic likely better than malicious code execution

HLL would prevent kernel exploits

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

- 1 Is BISCUIT's performance roughly similar to Linux?
- 2 What is the breakdown of HLL tax?
- 3 How much might GC cost?
- 4 What are the GC pauses?
- 5 What is the performance cost of Go compared to C?
- 6 Does BISCUIT's performance scale with cores?

## Experimental setup

### Hardware:

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

### Eval applications:

- 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

Ran for a minute

512MB heap RAM for BISCUIT

## 1: Is BISCUIT's perf roughly similar to Linux?

i.e. is BISCUIT's performance similar to production-grade kernel?

Compare app throughput on BISCUIT and Linux

Debian 9.4, Linux 4.9.82

Disabled features that slowed Linux down on our apps:

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



## 1: Is BISCUIT's perf roughly similar to Linux?

	<b>BISCUIT ops/s</b>	<b>Linux ops/s</b>	<b>Ratio</b>
CMailbench (mem)	15,862	17,034	1.??
NGINX	88,592	94,492	1.??
Redis	711,792	775,317	1.??

Linux has more features: NUMA, scales to many cores, ...

Not apples-to-apples, but BISCUIT perf roughly similar

## 2: What is the breakdown of HLL tax?

Record CPU time profile of our apps

Categorize samples into HLL cost buckets

Measure HLL tax of our apps:

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

## Prologue cycles are most expensive

	<b>GC cycles</b>	<b>GCs</b>	<b>Prologue cycles</b>	<b>Write barrier cycles</b>	<b>Safety cycles</b>
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 few long-lived kernel heap objects

## GC cost varies by program

More live data  $\implies$  more cycles per GC

Less free heap RAM  $\implies$  GC more frequent

Total GC cost  $\propto$  ratio of live data to free heap RAM

### 3: How much might GC cost?

Created two million vnodes of live data

Varied free heap RAM

Ran CMailbench, measured GC cost

### 3: How much might GC cost?

Live (MB)	Free (MB)	Ratio	Tput	GC%
640	320	2	10,448	34%
640	640	1	12,848	19%
640	1280	0.5	14,430	9%

⇒ Need 3× heap RAM to keep GC < 10%

### 3: GC memory cost in practice?

Few programs allocate millions of resources

MIT's big time-sharing machines:

80 users, 800 tasks, 9-16GB RSS, <2GB kernel heap

(Exception: cached files, maybe evictable)

Memory cost acceptable in common situations?

## GC pauses

GC must eventually execute

Could delay latency-sensitive work

Some GCs cause one large pause, but not Go's

- Go's GC is interleaved with execution (*Baker'78, McCloskey'08*)
- Causes many small delays



## 4: What are the GC pauses?

Measured duration of each GC pause during NGINX

Multiple pauses occur during a single request

Sum pause durations over each request

## 4: What are the GC pauses?

Max single pause: 115  $\mu$ s

(marking large part of TCP connection table)

Max total pauses during request: 582  $\mu$ s

Less than 0.3% of requests paused  $> 100\mu$ s

## 4: GC pauses OK?

Some programs can't tolerate rare 582  $\mu$ s pauses

But many probably can

99%-ile latency in service of Google's "Tail at Scale" was 10ms

## 5: What is the cost of Go compared to C?

Compared OS code paths with identical functionality

Chose paths that are:

- core OS paths
- small enough to make them have same functionality

Two code paths in *OSDI'18* paper

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

## 5: What is the cost of Go compared to C?

Pipe ping-pong code path:

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

## 5: C is 15% faster

Pipe ping-pong:

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

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

Go slower, but competitive

## 6: Does BISCUIT scale?

Can BISCUIT efficiently use many cores?

Is Go scalability bottleneck?

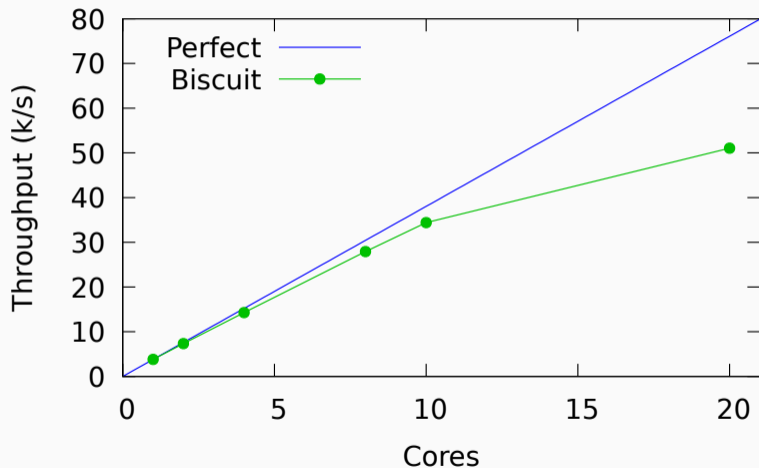
## 6: Does BISCUIT scale?

Ran CMailbench, varied cores from 1 to 20

Measured throughput



## 6: BISCUIT scales well to 10 cores



Lock contention in CMailbench at 20 cores, not NUMA-aware

## 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)

## 6.1810 and HLL

Should we use HLL in 6.1810?

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