CS 134
Operating Systems

April 10, 2019

Biscuit

Motivation

- Commodity kernels are written in C
- For good reason: C gives programmer total control

But, C is hard to use correctly

- Memory management left to the programmer
- Serious problems left for kernel developers
  - Concurrent data structures challenging (RCU, next week)
  - Memory safety bugs
    - Use-after-free (difficult to debug)
    - Buffer overflows (security vulnerabilities)
    - https://source.codeaurora.org/quic/la//kernel/msm-3.14/commit/?id=72f67b29a9c5e6e8d3c34751600c749c5f5e13e1
    - CVE-2017-0619
    - https://source.codeaurora.org/quic/la//kernel/msm-3.10/commit/?id=9656e2c2b3523af20502bf1e933e35a397f5e82f

HLL automatically eliminates memory safety bugs

- HLLs have a garbage collector (GC)
- GC automates memory deallocation
- Convenient for programmer
  - and provides memory safety
- But, GC has costs:
  - CPU cycles at runtime
  - Delays execution
  - Extra memory
- HLL is a tradeoff: safety (and ease-of-use) vs. performance

HLL automatically eliminates memory safety bugs

- Determining performance cost is important to understand the tradeoff
- No in-depth performance evaluation of HLL kernels has been done before
  - Despite researchers building many HLL kernels
- Want: better understanding of HLL kernel perf
- Goal: Compare performance of HLL kernel against fast kernel
  - Against Linux

Biscuit

- Started in 2014
  - 30K LOC in Go
- Architecture similar to Linux (for fair comparison)
- POSIX system calls
  - Can run complex apps (like Redis and NGINX)

Some differences: Biscuit vs. Linux

- Kernel threads are light-weight *goroutines*
  - Context switch in kernel doesn't save/restore page table
  - Cannot dereference user pointers
    - Manual translation
- Go isn't designed to handle interrupts
  - Runtime doesn't enable/disable interrupts during critical sections
    - Calling critical sections (e.g., allocation) during interrupt handler could deadlock
  - Interrupt handler can't do much
  - Instead, they wakeup a handling goroutine and return
- Biggest difference: handling OOM

Out of memory (OOM)

- Problem Biscuit, Windows, Linux, FreeBSD all face
- Many kernel operations allocate memory
  - open(2) allocates a file object
  - socket(2) allocates a socket object
- User program decides when to release the resource/free the mem
- →User program controls how much heap memory the kernel uses
- But, machine has limited memory

Problem: what if user code cause kernel to allocate all memory?

- Why would this happen?
  - Buggy program
  - Database server is intentionally using most memory
  - Unlucky spike in allocations
- Result: almost no operation can succeed until memory is freed
  - Hard or impossible for user program to handle sensibly
  - e..g, printf(3) fails
  - exit(2) fails!
- No user program can make progress

How to recover? Need to free memory

- Linux's approach
  - Blocking in allocator is tempting
    - Then, caller doesn't have to handle failure
  - But, this can deadlock
    - E.g., Good program takes lock on directory in FS
      - Memory hog is waiting for the lock
      - Kernel can't kill hog
      - Hog waits for good program
      - Good program waits for hog to exit (freeing mem)
      - Deadlock
    - Avoid deadlock by failing alloc of good program
  - Kernel must handle failure of nearly all allocations
    - Hard and filled with bugs
    - Unwritten "rule": Too small to fail

Biscuit's approach

- Can't use Linux approach
- Before executing op, wait until enough mem
  - No waiting in the middle of an op
  - No locks held
  - Thus, no deadlock
- How to calculate max mem
  - Static analysis of Go is easy
  - Tool: MAXLIVE
    - High level: fancy escape analysis
    - Not exact, but conservative
    -

Tool: MAXLIVE

- High level: fancy escape analysis
- Not exact, but conservative
- Inspects call graph
- Finds all allocations at each syscall
- Two kinds of objects:
  - 1) May be written to global
  - 2) Only ever referenced by stack pointer
- Type 1 objects always live
- Type 2 objects freed on some stack frame destruction
- Max mem = sum of type 1 + max of type 2 at each call graph leaf
- Result: no deadlock, almost no handling allocation failures

Experience

- 90 uses of unsafe (casting pointers, etc.)
- Hacked the runtime in a couple of ways:
  - Schedule interrupt goroutines
  - Count allocations
- Go was helpful
  - Slices vs. pointer + size
  - Defer vs. goto
  - Closures
  - Maps
  - GC vs. manual memory management
    - Significantly simplifies concurrent data structures
    - Entries heap allocated
    - When to free an entry?
    - When are all other threads done with an entry?

```
f := createFile("/tmp/defer.txt")
defer closeFile(f)
writeFile(f)
```

Experience

- Implemented many other optimizations to compete with Linux:
  - Map kernel text with large pages
  - Per-CPU NIC TX queues
  - Directory cache with lock-free lookup (RCU)
  - Go didn’t prevent their implementation

Performance

- Three demanding applications
  - NGINX: webserver
  - Redis: key-value store
  - Cmailbench: fork/exec/VM benchmark
- Exercise 10GB NIC, TCP stack, VM, FS
- No idle CPU cycles
- At least 80% of CPU time spent in kernel

Results

- Linux comparison
  - Is Biscuit performance in the same league?
  - Disabled expensive Linux features
    - Speeds Linux up
    - Makes comparison fair
  - Biscuit within 9% of Linux on our test apps

Results: GC

- GC < 3%
  - Cost of GC determined by two factors:
  1. Number of objects
     - GC must mark/read pointers in each object
  2. Amount of free heap in memory
     - GC each time free memory exhausted
  - Apps use up to 5GB memory
  - And cause kernel to allocate rapidly
  - But # of kernel objects is small
    - Kernel heap contains small metadata objects
    - Separate allocator for pages
      - User memory, file content, socket buffers, page-table pages
    - Reduces # kernel objects. Increases heap free mem

Results

- Isolate performance differences due to high-level language from diff OS features
  - Modified Linux/Biscuit to get two nearly identical code paths
    - Pipe ping-pong
    - Page fault
  - CPU-time profiles show both OSes doing the same thing
  - Ping-pong 15% faster
    - Go version has safety checks/write barriers
  - Page fault 5% faster
    - Kernel entry/exit and copying dominate other work

Results

- GC pauses
  - During GC, each allocation must do complete GC work
  - Pauses come from GC work

  - Max single pause of 115 μs
  - Pauses can accumulate in a system call
  - Max accumulated during tests: 574 μs

  - Pause times were reduced by tuning GC

Conclusion

- High-level language worked pretty well
- Performance is pretty good
- But, C is faster

Questions

- What about other languages, like Python?
  - Other HLLs would likely have much different performance results
  - Multithreading in Python is harder than in Go

- How does Biscuit access physical addresses without pointer arithmetic?
  - Uses “unsafe” pointer conversion

- Why is nearly-identical Go code slower than C?
  - The Go compiler inserts more instructions
  - Safety-checks (e.g., array access)
  - Write barriers

Questions

- Does a reliance on different Go packages pose a larger challenge when attempting to scale?
  - Seems more like a fixed overhead cost
- Does a HLL like Go increase the frequency or number of runtime errors that can occur
  - Yes. Fail fast, fail often. Better to fail than silently allow corruption or exploitation
- Doesn’t waiting (for heap space) in system calls bottleneck performance?
  - The alternative is possibly worse (OOM killer)

Questions

- What features make Go statically analyzable and not C?
  - Go is a simple language to parse. Most importantly, `go` package includes: scanner, parser that'll generate an Abstract Syntax Tree (AST).
- Would less access to hardware cause some bugs to be harder to debug?
  - Unclear that Go has less access to hardware.
- How does Go ensure type-safety
  - Other than `unsafe` package, no way to mess around with types and crash (no direct access to raw pointers).
- How does Linux deal with heap exhaustion?
  - OOM killer kills process with high memory, low CPU

Questions

- Given safety, and easier development with Go, why are C kernels popular?
  - *Worse is better*: "Unix and C are the ultimate computer viruses"
- Why do they need a shim layer and how does it work?
  - Go runtime assumes it's running on an OS and makes system calls to: allocate memory and control go threads. Shim layer provides those features as kernel code.
- Does it matter to user or library author what language the kernel is written in?
  - No

Questions

- Why haven't there been any new low-level languages to replace C?
  - C fits its niche well
- How viable would it be to write an OS in Haskell?
  - Existence proof: House, Kinetic, hos
- For JOS, haven't had to dynamically allocate mem and later free it. How much of a problem is allocation/freeing in other OSes?
  - xv6: has a little allocation. Linux has much.