An analysis of a directory entry cache in a high level language

Robert Cunningham
mentored by Cody Cutler

Background and Motivation

What languages are kernels generally written in?

C

C

C

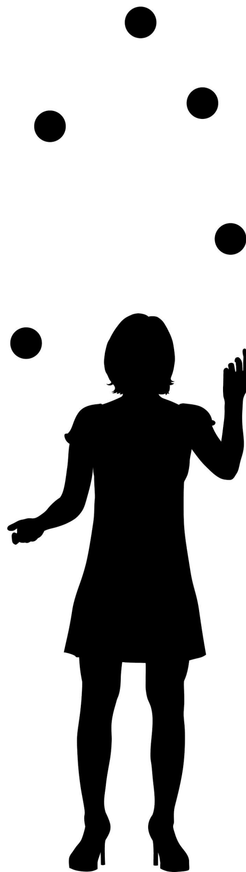
Why is C so popular for kernels?

What are the disadvantages of C?

- The programmer must manage a large number of low level tasks and subsystems.

Examples of low level tasks?

- Memory management [malloc() and free()]
- Reference counting
- Pointer arithmetic
- etc.



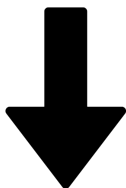
An analogy.

Should racecar drivers be concerned about whether the wheels are bolted onto the car tightly enough?

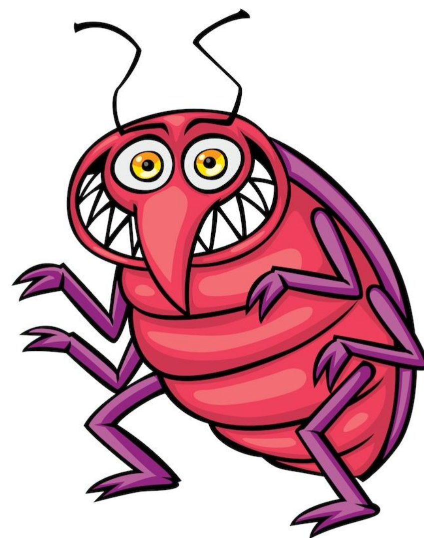
Of course not. We want them to focus on the big picture.

How does this apply to programmers?

Forcing too much overhead onto a programmer, such as by forcing them to manage low level subsystems (as C does), can cause the creation of subtle bugs.



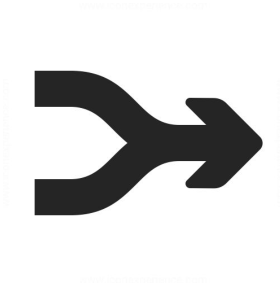
This can result in crashes, incorrect behavior, and security vulnerabilities.

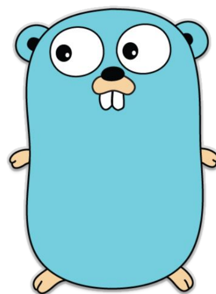


How can we reduce complexity?

Let’s pick a new language… that manages low level subsystems automatically!

However, it should also be pretty fast.





Let’s use **Go**!

How can we test whether this will reduce complexity?

We will build a small part of an operating system to test this theory: a directory entry cache (“dcache”).

What is a dcache?

What is a dcache?

A dcache, also known as a Directory Name Lookup Cache, is responsible for caching directory paths.

In short, it caches recently looked up paths to RAM, so they don’t take expensive disk reads.

This speed up matters a lot for programs like GNU Make, which do lots of concurrent path lookups.

Example of a file lookup without a dcache.

```
Have you heard of /home/?

No, let me check the disk.. .. .. .. .. .. ..yes, it exists.

Have you heard of /home/robert?

No, let me check the disk.. .. .. .. .. .. ..yes, it exists.

Have you heard of /home/robert/cats/?

No, let me check the disk.. .. .. .. ..no, does not exist.
```

Example of a file lookup with a dcache

Have you heard of /home/?

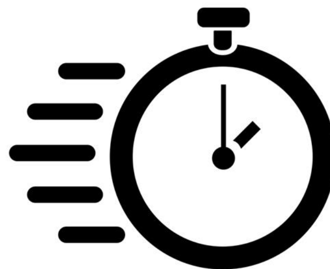
Yes (dcache hit).

Have you heard of /home/robert/?

Yes (dcache hit).

Have you heard of /home/robert/cats/?

No, sorry (dcache miss). Let me check the disk to be sure..
.. .. .. Sorry, it does not exist.

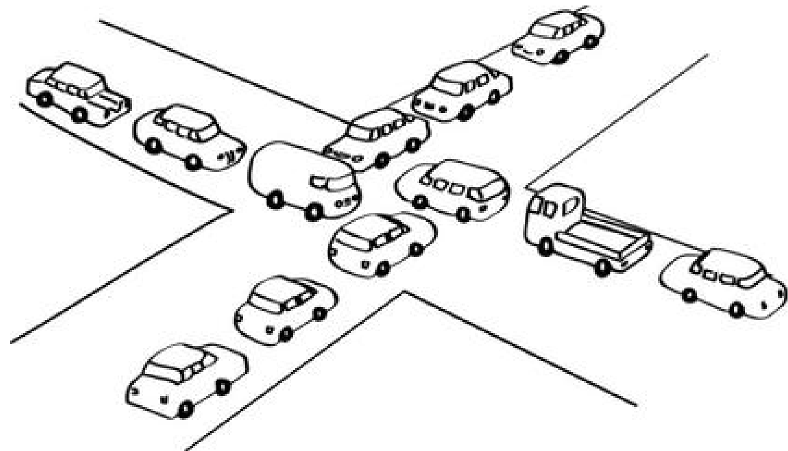


Concurrency in dcaches

It matters a lot that our dcache is capable of supporting requests concurrently and correctly.

Sequential path lookup would be a crippling performance hit.

However, this is nontrivial to implement.



A quick “history” lesson

- Until 2.5.10, Linux used a single global *dcache_lock* lock to control concurrent access to the dcache
  - Can barely be considered concurrent.
  - Correct, but slow.

- In 2.5.10, this global lock was replaced with a system of seqlocks.
  - Significantly better, but lots of reading and writing locks involved, which isn’t super fast.

- In 2.5.62, we got the modern system that we have today.
  - In most cases avoids reading and writing a lot of locks.

What is a path walking algorithm?

A path walking algorithm is responsible for taking paths like

/home/robert/cats/fluffy.jpg

and querying the dcache for the corresponding dentries. Modern dcaches use two such algorithms: RCU walk and REF walk.

How do state of the art dcaches manage concurrency?

2 separate path walking algorithms.

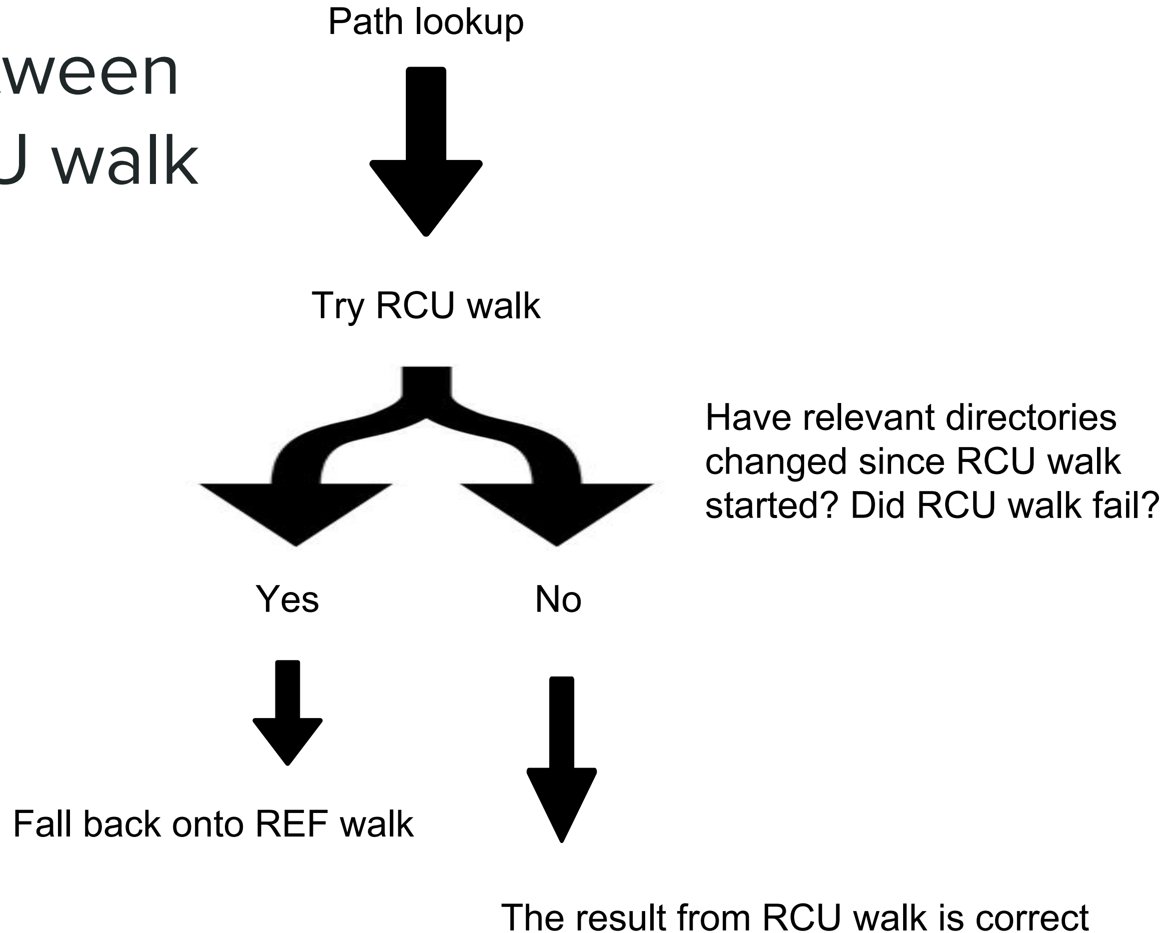
REF walk

- Takes locks
- Guaranteed to respond yes or no.
- Can not run concurrently with itself on the same directories
- Slow and safe.

RCU walk

- “Invisible:” takes no locks and doesn’t write to disk
- Might not find an answer
- Can run concurrently with itself on the same directories
- Fast and risky.

Interplay between REF and RCU walk



How does RCU walk check whether relevant directories have changed?

Seqlocks

What is a seqlock?

- Seqlocks allow us to check that data hasn't changed between when some code begins and ends.

```
type SeqLock struct {
    count uint32
    writeLock SpinLock
}

retry:
start := seqlock.getCount()
try()
if seqlock.getCount() != start {
    goto retry
}
```

```
func (s *SeqLock) write_seqlock() {
    s.writeLock.Lock()
    atomic.AddUint32(&s.count, 1)
}

func (s *SeqLock) write_sequnlock() {
    atomic.AddUint32(&s.count, 1)
    s.writeLock.Unlock()
}
```

Can we simplify the implementation of a dcache using Go?

Yes.

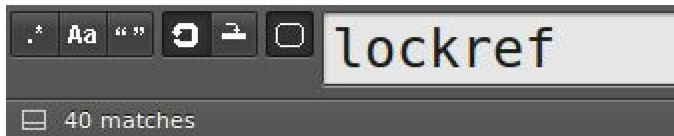
The short answer

Comparison of C and Go for dcaches

Keeping track of which memory is in use

C

- Each dentry has a special counter, called a *lockref*, which keeps track of how many references to a given dentry exist
- Whenever the programmer takes a new reference, he or she must remember to update the lockref.
- From time to time the programmer explicitly checks whether the lockref is at 0



Go

- The built-in garbage collector manages everything automatically.

```
// Automatically managed in Go
```

Ensuring that memory changes appear atomic

C

- A complex system, *Read Copy Update*, provides an API for ensuring that all threads execute a context switch before any shared memory is freed
- The programmer must recognize and understand the problem cases and call the RCU functions appropriately

```
rcu_read_lock();
//...
rcu_read_unlock();
```

Go

- The built-in garbage collector manages everything automatically.

```
// Automatically managed in Go
```

String support

C

- Use of pointer arithmetic and keeping various custom-made structs so string length is easy to pass around.

```
struct qstr {
    union {
        struct {
            HASH_LEN_DECLARE;
        };
        u64 hash_len;
    };
    const unsigned char *name;
};
```

Go

- Strings are supported natively.

```
"No structs required"
```

String processing support

C

- Custom, error prone implementations of even the most basic functions like string compares.

```c
static inline int dentry_string_cmp(const unsigned char *cs, const unsigned char *ct, unsigned tcount)
{
    unsigned long a,b,mask;

    for (;;) {
        a = *(unsigned long *)cs;
        b = load_unaligned_zeropad(ct);
        if (tcount < sizeof(unsigned long))
            break;
        if (unlikely(a != b))
            return 1;
        cs += sizeof(unsigned long);
        ct += sizeof(unsigned long);
        tcount -= sizeof(unsigned long);
        if (!tcount)
            return 0;
    }
    mask = bytemask_from_count(tcount);
    return unlikely(!!((a ^ b) & mask));
}
```

Go

- *strings* package contains correct implementations of many useful functions.

```go
if "Couldn't be" > "easier" {

}
```

Is there anywhere where Go falls short?

- Kind of. Go is lacking **convenient preprocessor macros**, which are opaque, but very useful and highly readable.

```
* container_of - cast a member of a structure out to the containing structure
* @ptr:    the pointer to the member.
* @type:   the type of the container struct this is embedded in.
* @member: the name of the member within the struct.
```

*Go technically does have a container_of implementation, but it runs counter to design principles of Go (clarity and readability), so we choose not to use it.

Custom loops

C

- C has convenient preprocessor macros, which are concise and highly readable, if opaque.

```
hlist_bl_for_each_entry_rcu(dentry, node, b, d_hash) {
```

Go

- Go's design principles encourage transparency, which can sometimes get ugly.

```
for {
    if(...) {
        goto next
    }

    next:
    node = node.next
    if node == nil {
        return nil
    }
    dentry = node.data.(*Dentry)
}
```

Usefulness of container_of

C

- List nodes only hold pointers, and we can move outwards to their enclosing type when we've found the one we want using container_of.

```
struct hlist_node {
    struct hlist_node *next, **pprev;
};
```

```
container_of(...)
```

Go

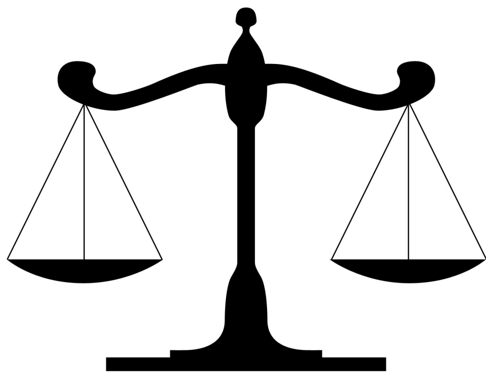
- All list nodes must contain an ugly uncast reference to their parent datum, which we then cast back upon retrieval.

```
type list_node struct {
    next *list_node
    prev *list_node
    data interface{}
}
```

```
dentry = node.data.(*Dentry)
```

Are macros actually a simplification?

- Macros abstract away complexity.
- This can be highly useful, since it makes the code easily human readable.
- Serious risk of creating bugs through subtle misunderstanding or misuse.



Implementation Progress

Current status

Right now we have a functional single threaded dcache whose concurrency support is in progress.

Path resolution

```
---------------------------
|-kom
  |-jpk
|-opc
  |-lbl
|-mnu
  |-dpa
  |-wxt
    |-mdp
  |-zxc
Correctly resolved /opc/lbl.
Correctly resolved /././//./kom/../kom/jpk.
Correctly resolved /mnu/../mnu/dpa/../wxt/mdp/../mdp/../../dpa/./../zxc/../zxc.
Correctly resolved /kom/jpk.
Correctly resolved /..
Correctly resolved /kom/jpk.
Correctly resolved /kom/jpk///../jpk.
Correctly resolved /./kom/jpk/../jpk/../jpk///./../jpk.
Correctly resolved /kom/jpk/../jpk///../jpk/./kda/../jpk/../jpk/../../mnu/zxc/./../././..///mnu/../mnu/../opc/lbl.
Correctly resolved /etd/kom/..///opc/lbl///../../kom/jpk/../jpk///../jpk/../jpk/../jpk/../jpk/././../jpk/fsv/../jpk/../jpk/ozq/../jpk/vys///../jpk/..
```

File system API

```
Added /myDir
CURRENT STATE OF FILESYSTEM
---------------------------
|-myDir


Added /myDir/myFile
CURRENT STATE OF FILESYSTEM
---------------------------
|-myDir
  |-myFile


Opened /myDir/myFile, which had data myFileData


Deleted /myDir/myFile
DENTRY: [Dentry; Name: myFile, Parent: myDir]
CURRENT STATE OF FILESYSTEM
---------------------------
|-myDir
  |-myFile  [NEGATIVE]
```

General infrastructure

- Time consuming but important.
- Go implementation of spinlocks, seqlocks, linked lists, locking lists, striped maps, etc.

Conclusions

Conclusions

The combination of a garbage collector and other conventional of high level languages makes implementing a dcache in Go much simpler.

Although Go doesn’t support convenient macros, whether they actually offer a simplification is up for debate.

Future work

Ideally we'd finish the Go implementation of a concurrent dcache so we can quantify the performance implications of using a Go implementation versus a C dcache implementation.

Acknowledgements

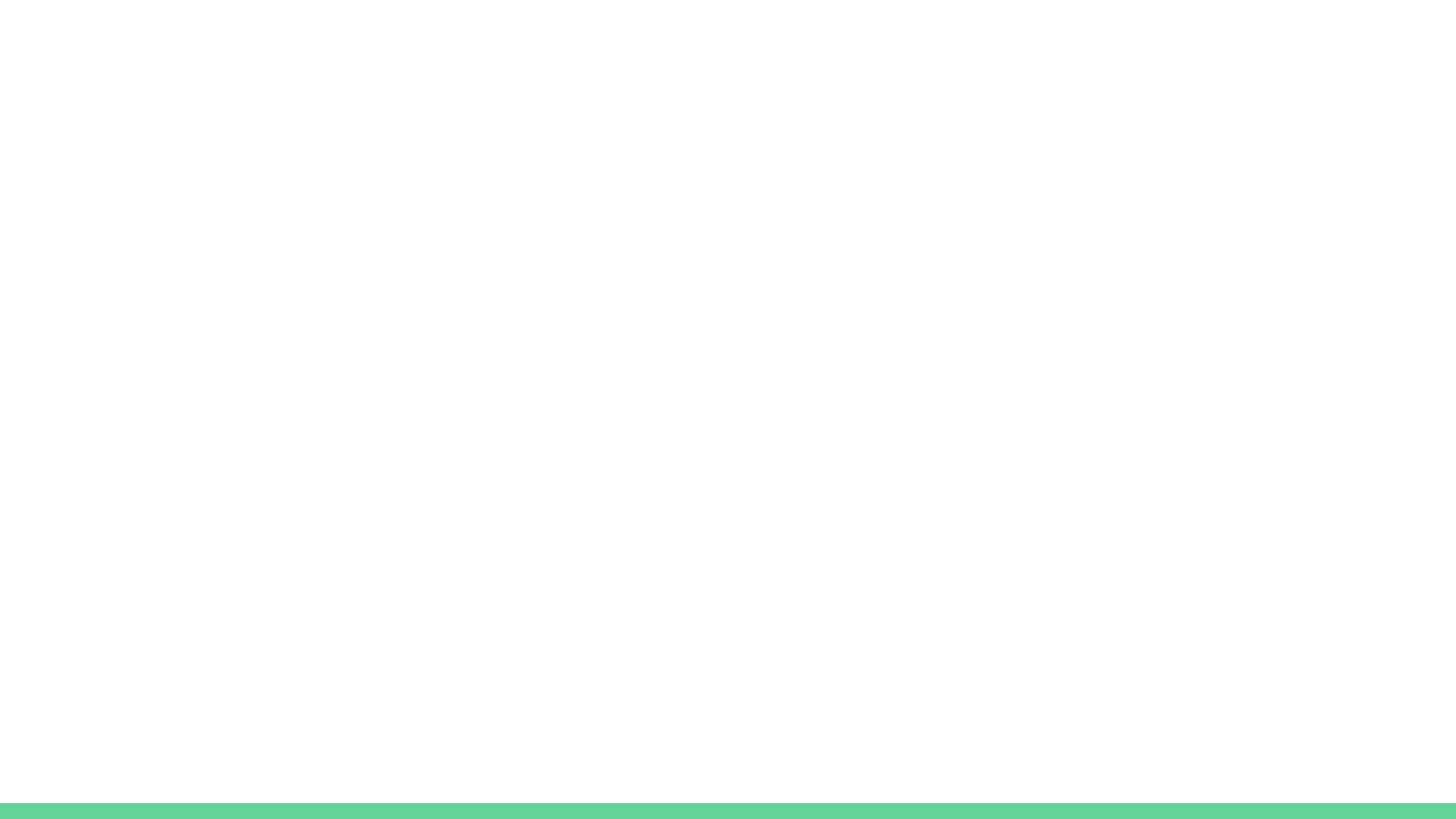
None of this would be possible without...

- Cody Cutler
- MIT PRIMES
- Professor Kaashoek
- My parents

That’s all.

Questions?

Thanks for listening!



question

C

- a

Go

- b