Concurrency in Go

"Don't communicate by sharing memory; instead, share memory by communicating"

Matt Holiday

3 April 2019

Cardinal Peak

What's the problem?

I want to find duplicate files based on their **content**

What's the problem?

I want to find duplicate files based on their **content**

Use a secure hash, because the names / dates may differ

```
f088913 2
    /Users/mholiday/Dropbox/Emergency/FEMA_P-320_2014_508.pdf
    /Users/mholiday/Dropbox/Emergency/nps61-072915-01.pdf
```

What's the problem?

I want to find duplicate files based on their **content**

Use a secure hash, because the names / dates may differ

```
f088913 2
    /Users/mholiday/Dropbox/Emergency/FEMA_P-320_2014_508.pdf
    /Users/mholiday/Dropbox/Emergency/nps61-072915-01.pdf
```

It takes nearly **5 minutes** to comb through my Dropbox folder

Sequential Approach

How it works: Declarations

```
package main

import (
    "crypto/md5"
    "fmt"
    "io"
    "log"
    "os"
    "path/filepath"
)

type pair struct {
    hash string
    path string
}

type fileList []string
type results  map[string]fileList
```

How it works: Hashing

```
func hashFile(path string) pair {
    file, err := os.Open(path)

    if err != nil && err != os.ErrNotExist {
        log.Fatal(err)
    }

    defer file.Close()

    hash := md5.New() // fast & good enough

    if _, err := io.Copy(hash, file); err != nil {
        log.Fatal(err)
    }

    // make a string so we can use it as a map key

    return pair{fmt.Sprintf("%x", hash.Sum(nil)), path}
}
```

How it works: Searching

```
func searchTree(dir string) (results, error) {
    hashes := make(results)

    err := filepath.Walk(dir, func(p string, fi os.FileInfo,
                                   err error) error {

        // ignore the error parm for now

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            h := hashFile(p)
            hashes[h.hash] = append(hashes[h.hash], h.path)
        }

        return nil
    })

    return hashes, err
}
```

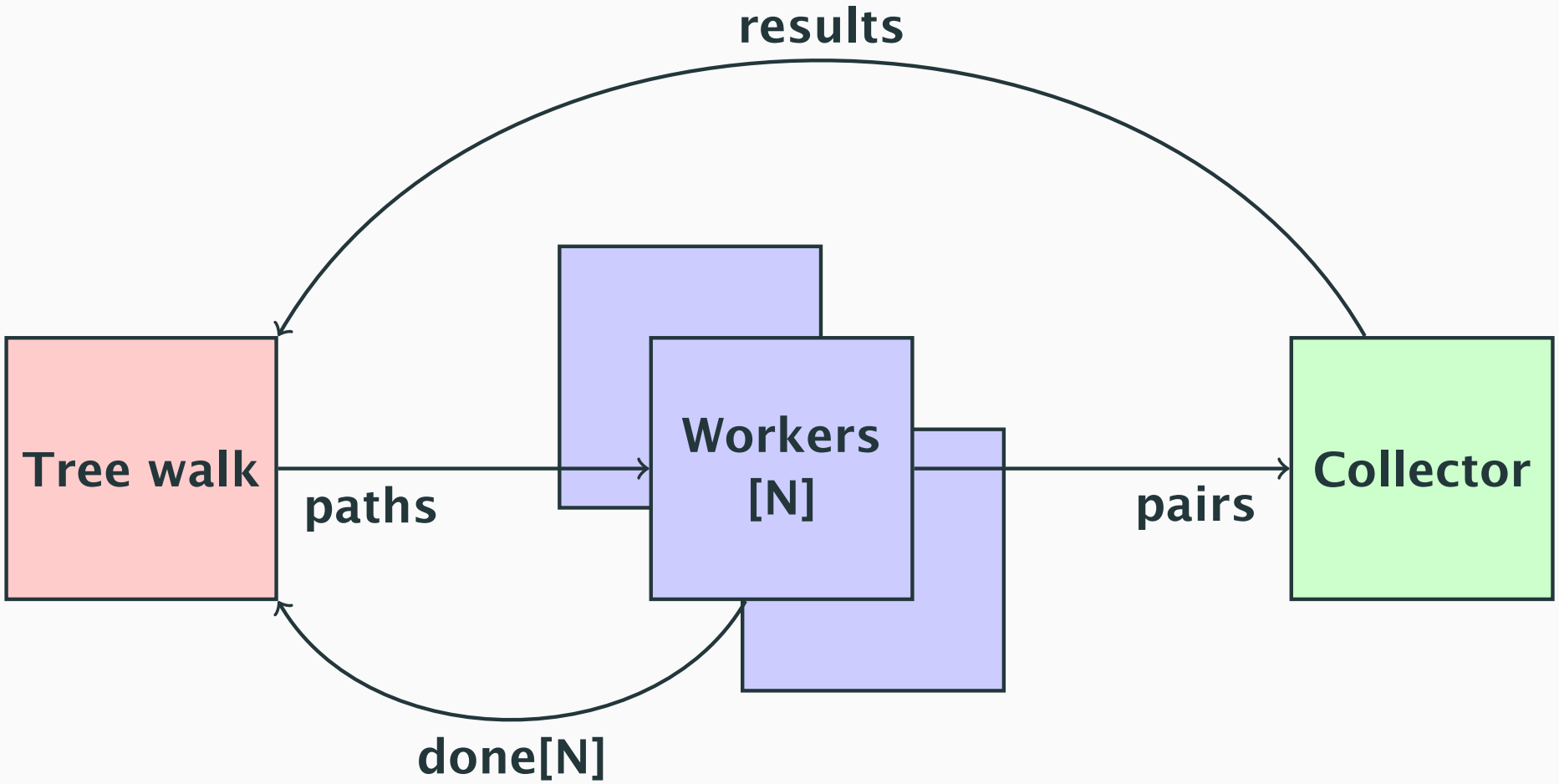
How it works: Output

```
func main() {
    if len(os.Args) < 2 {
        log.Fatal("Missing parameter, provide dir name!")
    }

    if hashes, err := searchTree(os.Args[1]); err == nil {
        for hash, files := range hashes {
            if (len(files) > 1) {
                // we will use just 7 chars like git

                fmt.Println(hash[:7], len(files))

                for _, file := range files {
                    fmt.Println("   ", file)
                }
            }
        }
    }
}
```

Concurrent Approach #1

A concurrent approach (like map-reduce)

Use a fixed pool of goroutines and a collector and channels



How it works: Collecting the hashes

```
func collectHashes(pairs <-chan pair, result chan<- results) {
    hashes := make(results)

    for p := range pairs {
        hashes[p.hash] = append(hashes[p.hash], p.path)
    }

    result <- hashes
}
```

How it works: Replacing the processor

```
func processFiles(paths <-chan string, pairs chan<- pair,
                  done chan<- bool) {
    for path := range paths {
        pairs <- hashFile(path)
    }

    done <- true
}
```

How it works: Replacing the tree walk

```
workers := 2 * runtime.GOMAXPROCS(0)

paths := make(chan string)
pairs := make(chan pair)
done := make(chan bool)
result := make(chan results)

for i := 0; i < workers; i++ {
    go processFiles(paths, pairs, done)
}

// we need another goroutine so we don't block here

go collectHashes(pairs, result)

. . .
```

How it works: Replacing the tree walk

```
err := filepath.Walk(dir, func(p string, fi os.FileInfo,
                                err error) error {
    // again, ignore the error passed in

    if fi.Mode().IsRegular() && fi.Size() > 0 {
        paths <- p
    }

    return nil
})

if err != nil {
    log.Fatal(err)
}

// we must close the paths channel so the workers stop
close(paths)
. . .
```

How it works: Replacing the tree walk

```
. . .

// wait for all the workers to be done

for i := 0; i < workers; i++ {
    <-done
}

// by closing pairs we signal that all the hashes
// have been collected; we have to do it here AFTER
// all the workers are done

close(pairs)

hashes := <-result

return hashes
```

Evaluation #1

56.11s in the version shown above

Evaluation #1

56.11s in the version shown above

52.76s with a buffer to `pairs`
(buffering `pairs` keeps the workers working)

Evaluation #1

56.11s in the version shown above

52.76s with a buffer to `pairs`
(buffering `pairs` keeps the workers working)

51.36s with twice as many workers

Concurrent Approach #2

Another concurrent approach

Add a goroutine for each directory in the tree

Another concurrent approach

Add a goroutine for each directory in the tree

This improves the performance slightly, we're not waiting on paths to be identified

Another concurrent approach

Add a goroutine for each directory in the tree

This improves the performance slightly, we're not waiting on paths to be identified

51.14s in the basic version

Another concurrent approach

Add a goroutine for each directory in the tree

This improves the performance slightly, we're not waiting on paths to be identified

51.14s in the basic version

50.03 adding buffers on channels to/from workers

Another concurrent approach

Add a goroutine for each directory in the tree

This improves the performance slightly, we're not waiting on paths to be identified

51.14s in the basic version

50.03 adding buffers on channels to/from workers

48.75 with twice as many workers

How it works: Parallel tree walk

```
. . .

wg := new(sync.WaitGroup)

// multi-threaded walk of the directory tree; we need a
// waitGroup because we don't know how many to wait for

wg.Add(1)
err := walkDir(dir, paths, wg)

if err != nil {
    log.Fatal(err)
}

wg.Wait()
close(paths)

. . .
```

How it works: Parallel tree walk

```
func walkDir(dir string, paths chan<- string,
             wg *sync.WaitGroup) error {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, err error) error {
        // ignore the error passed in

        // ignore dir itself to avoid an infinite loop!
        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, paths)
            return filepath.SkipDir
        }

        . . .
```

How it works: Parallel tree walk

```
        . . .

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            paths <- p
        }

        return nil
    }

    return filepath.Walk(dir, visit)
}
```

Concurrent Approach #3

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

Without some controls, we'll run out of threads!

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

Without some controls, we'll run out of threads!

We'll limit the number of **active** goroutines instead

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

Without some controls, we'll run out of threads!

We'll limit the number of **active** goroutines instead

46.93s using 32 workers was the best time

The final approach: Goroutines galore!

Use a goroutine for every directory and file hash

Without some controls, we'll run out of threads!

We'll limit the number of **active** goroutines instead

46.93s using 32 workers was the best time

Adding more workers actually makes the time grow longer

Channels as counting semaphores

A goroutine can’t proceed without sending on the channel

Channels as counting semaphores

A goroutine can’t proceed without sending on the channel

The buffer provides a fixed upper bound (unlike a WaitGroup)

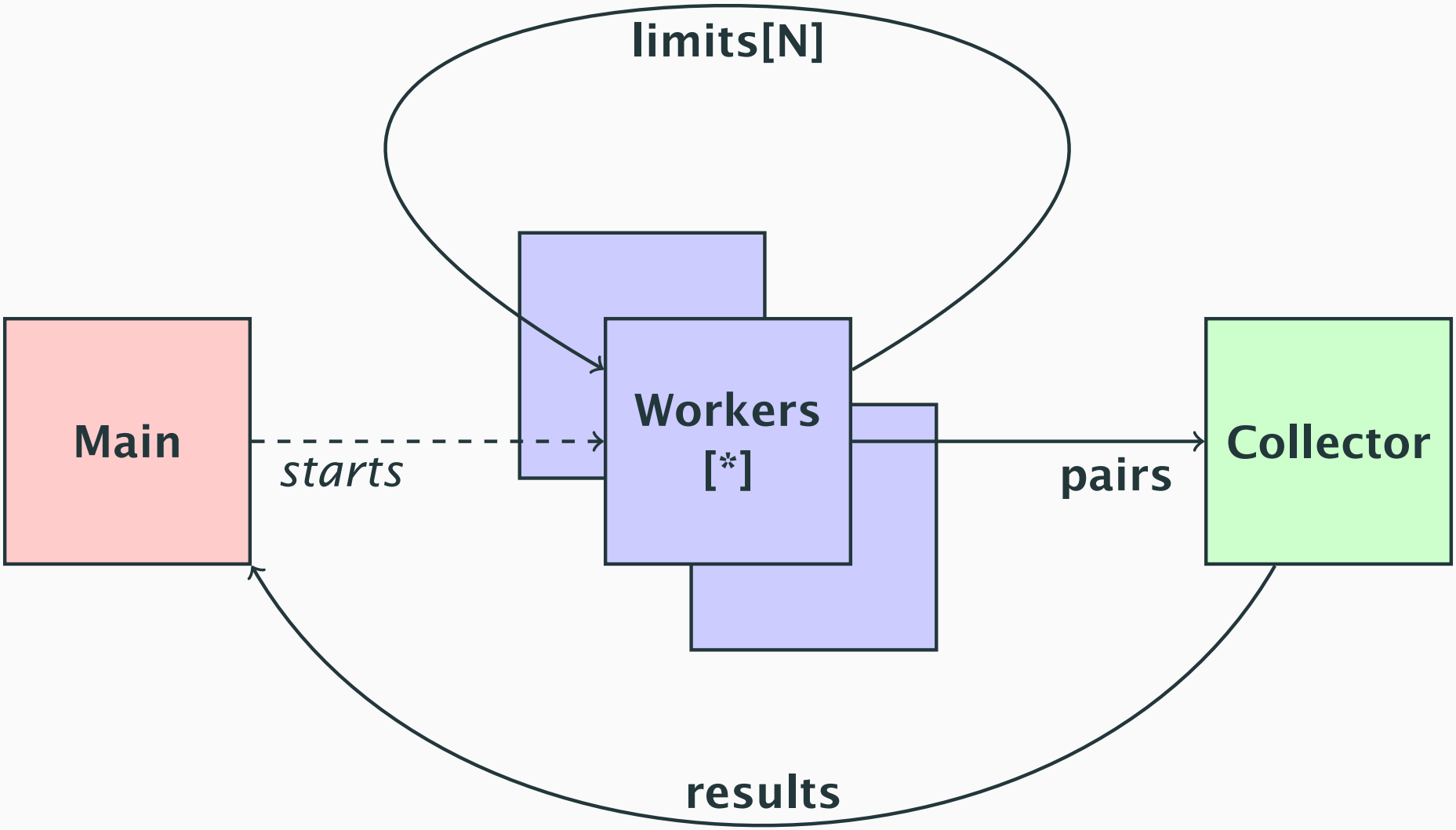
Channels as counting semaphores

A goroutine can’t proceed without sending on the channel

The buffer provides a fixed upper bound (unlike a WaitGroup)

One goroutine can start for each one that quits

What that looks like



How it works: Limiting goroutines

```
// we don't need a channel for paths or to signal done but
// we need a buffered channel to act as a counting semaphore

wg := new(sync.WaitGroup)
limits := make(chan bool, workers)
pairs := make(chan pair, workers)
result := make(chan results)

go collect(pairs, result)

. . .
```

How it works: Limiting goroutines

```
. . .

wg.Add(1)
err := walkDir(dir, pairs, wg, limits)

if err != nil {
    log.Fatal(err)
}

wg.Wait()
close(pairs)

hashes := <-result

return hashes
```

How it works: Modified processing

```
func processFile(path string, pairs chan<- pair,
                 wg *sync.WaitGroup, limits chan bool) {
    defer wg.Done()

    limits <- true

    defer func() {
        <-limits
    }()

    pairs <- hashFile(path)
}
```

How it works: Modified tree walk

```
func walkDir(dir string, pairs chan<- pair, wg *sync.WaitGroup,
             limits chan bool) error {
    defer wg.Done()

    visit := func(p string, fi os.FileInfo, err error) error {
        // ignore the error passed in

        if fi.Mode().IsDir() && p != dir {
            wg.Add(1)
            go walkDir(p, pairs, wg, limits)
            return filepath.SkipDir
        }

        . . .
```

How it works: Modified tree walk

```
        . . .

        if fi.Mode().IsRegular() && fi.Size() > 0 {
            wg.Add(1)
            go processFile(p, pairs, wg, limits)
        }

        return nil
    }

    limits <- true

    defer func() {
        <-limits
    }()

    return filepath.Walk(dir, visit)
}
```

Comparing Directories (extra)

Merkle trees: nodes with hashes

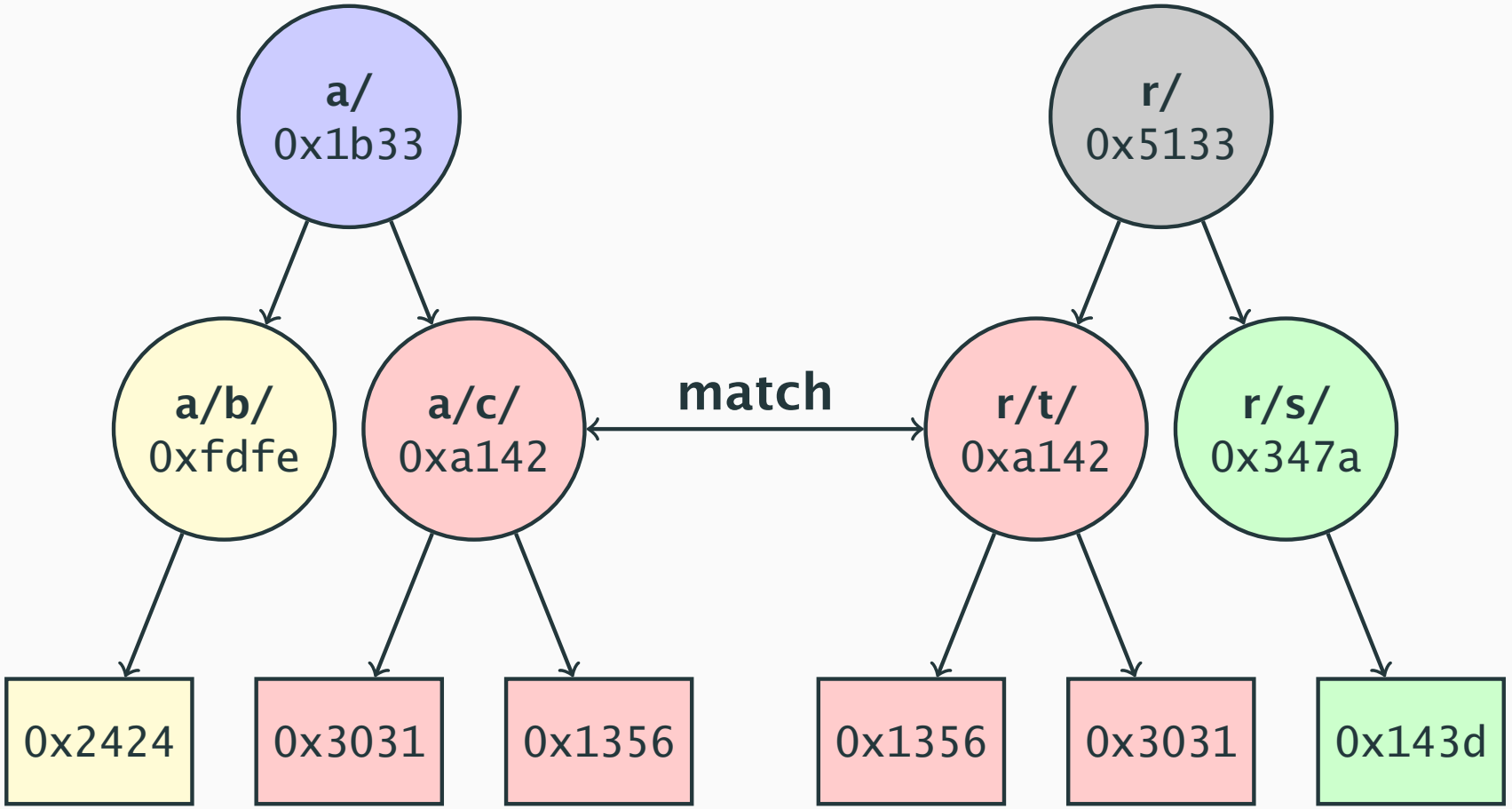
I'd like to identify entire directories that are identical to other directories (i.e., they have the same files, based on content, not name or mod date)

I can do that by building up a **Merkle tree**, which is a tree where each node has a hash based on its children

I'll calculate the hashes of files, and then calculate a hash over the collected hashes of a directory to get the hash for that directory as a whole

Merkle trees

Subtrees with the same hash are equal



How it works: Tree declarations

```
// one directory in the directory Merkle tree
type node struct {
    path  string
    hash  hash.Hash
    files []string
    child map[string]*node
}

// make a node to represent a directory in the tree
func makeNode(p string) *node {
    return &node{
        path:  p,
        hash:  md5.New(),
        child: make(map[string]*node)
    }
}
```

How it works: Modified collector

```
func collect(dir string, pairs <-chan pair,
             result chan<- results) {
    hashes := make(results)
    tree := makeNode(dir)

    for p := range pairs {
        hashes[p.hash] = append(hashes[p.hash], p.path)

        // insert this file hash into a merkle tree
        // using a path relative to the current dir

        rel, _ := filepath.Rel(dir, p.path)
        tree.insert(rel, p.hash)
    }

    tree.walk(hashes)  // and then calculate merkle hashes
    result <- hashes
}
```

How it works: Inserting

```
// insert a file's hash into the tree at the right spot
func (n *node) insert(path string, hash string) {
    s := strings.Split(path, "/")

    if len(s) == 1 {
        // we have a file in this directory
        n.files = append(n.files, hash)
    } else {
        // we may need to create the child on insert
        c := s[0]

        if n.child[c] == nil {
            n.child[c] = makeNode(filepath.Join(n.path, c))
        }

        n.child[c].insert(filepath.Join(s[1:]...), hash)
    }
}
```

How it works: Postorder traversal

```
// postorder walk to get a hash for each directory
func (n *node) walk(hashes results) string {
    for _, c := range n.child {
        n.files = append(n.files, c.walk(hashes))
    }

    // we need a consistent order to the hashes
    sort.Strings(n.files)

    for _, h := range n.files {
        n.hash.Write([]byte(h))
    }

    hash := fmt.Sprintf("%x", n.hash.Sum(nil))

    hashes[hash] = append(hashes[hash], n.path+"/")
    return hash
}
```

Some Gotchas

Gotchas 1: Deadlock

A goroutine is **preemptible** only when it starts a (non-inlined) function call, blocks on a channel or mutex, or makes a blocking system call.

Potential Problems

If your goroutine isn't preemptible, garbage collection will never run, because it must first "stop the world."

However, if the `main()` function exits, all goroutines terminate.

Gotchas 1: Deadlock example

```
go func() {
    var i byte

    for i = 0; i <= 255; i++ {
        // infinite loop does nothing
        // doesn't get elided
        // and can't be preempted
    }
}()

runtime.Gosched()    // yield execution
runtime.GC()         // force GC

// DEADLOCK

fmt.Println("Done")  // never happens
```

Gotchas 1: Deadlock fixed

```
go func() {
    var i byte

    for i = 0; i <= 255; i++ {
        // infinite loop does nothing
        // doesn't get elided

        // but we can yield
        if (i == 123) {
            runtime.Gosched()
        }
    }
}()

runtime.Gosched()    // yield execution
runtime.GC()         // force GC
fmt.Println("Done")  // prints Done
```

Gotchas 2: Closure capture

A closure shouldn't capture a **mutating** variable, e.g. a loop index.

If it does, it will get the wrong value!

```
for i := 0; i < 10; i++ {    // WRONG
    go func() {
        fmt.Println(i)
    }()
}
```

Instead, pass the variable's value as a parameter.

```
for i := 0; i < 10; i++ {  // RIGHT
    go func(i int) {
        fmt.Println(i)
    }(i)
}
```

Channel state reference

State	Receive	Send	Close
Nil	Block*	Block*	Panic
Empty	Block	Write	Close
Partly Full	Read	Write	Readable until empty
Full	Read	Block	
Closed	Default Value**	Panic	
Receive-only	OK	Compile Error	
Send-only	Compile Error	OK	

* `select` ignores a nil channel since it would always block

** Reading a closed channel returns `(<default-value>, !ok)`