Designing a Concurrent File Server

James Whitehead II

University of Oxford

jim.whitehead@cs.ox.ac.uk

28 August 2012

Building CSP Style Concurrent Systems

- Disciplined model of concurrency
- Build applications
  - Web server (CPA 2011)
  - File system server
- Range of designs

Go Programming Language

- Robert Griesemer, Rob Pike, Ken Thompson
- Fast compilation
- Lightweight type system
- Garbage collected
- Concurrent

Concurrency in Go

- Can FORK goroutines using ‘go’ keyword
- No PAR/FORKING block
- Any-to-any channels
- ALT using ‘select’
- Cites CSP (but not occam)

“Don’t communicate by sharing memory,
share memory by communicating”

File servers

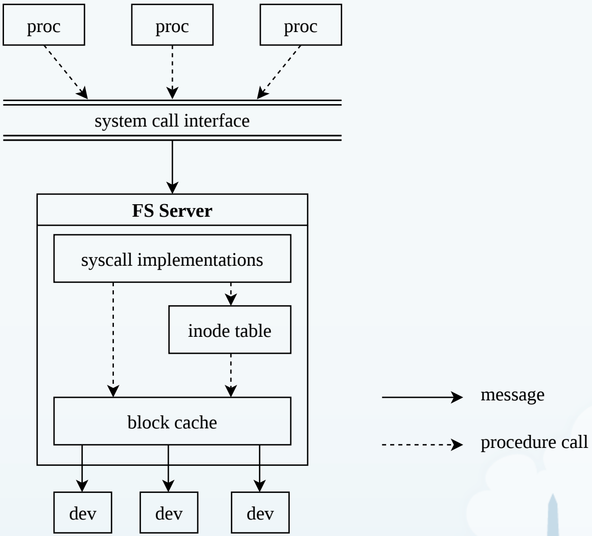
- Contention for resources
- Low-level I/O
- Shared data (disk)
- Shared memory
- Multiple users

File server concurrency

- Highly concurrent with explicit locks (UNIX, Linux)
- No concurrency (MINIX)
  - “Simple” source code
  - Embarrassingly sequential

MINIX provides a clean slate from which to derive the design of a concurrent file system.

Existing architecture

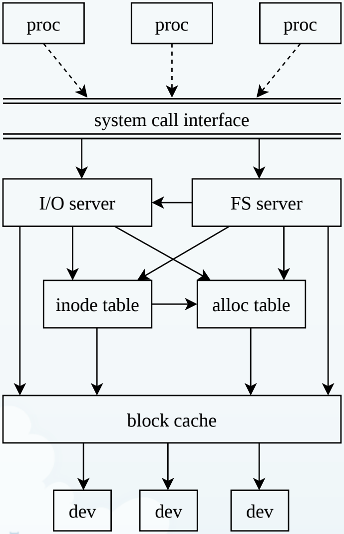


Opportunities for concurrency

- Files must be opened for I/O
- File descriptors
- Available until closed
- read, write, seek, etc.

An I/O operation concurrently with a normal system call.

A step in the right direction...



- New I/O server
- Explicit dependencies
- All sequential processes

Not free from hazards...

- Size of a file
- Updated in write (I/O)
- Read in stat (FS)

Data hiding/segregation

Trickle-down concurrency

Reduced the complexity of atomic actions, introducing three new bottlenecks.

- Block cache
- Inode table
- Alloc table

Block cache concurrency

- Cache-hit
  - Return the block immediately
- Cache-miss
  - Spawn new goroutine to load block
  - Continue serving other requests
  - Return block when ready

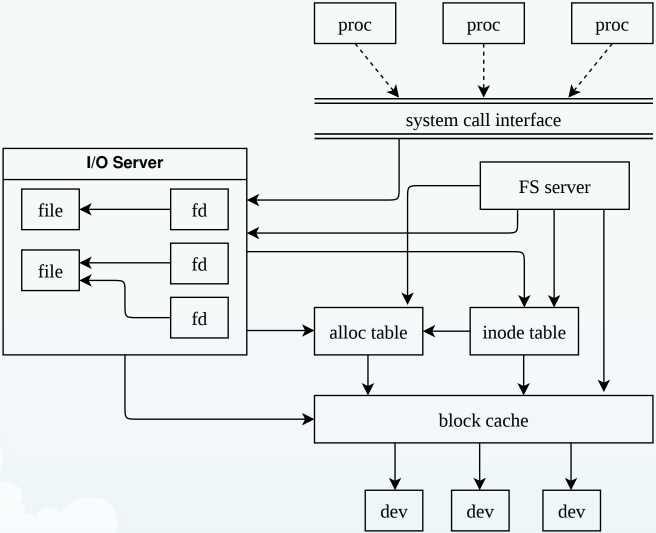
Slowly but surely..

Concurrent I/O Server

- Two open files are independent
- No shared blocks
- No shared inodes
- Allow CREW concurrency for each file

Shared file descriptors must be addressed..

A compromise...



Design

Overall:

- Process encapsulation
- Eliminate implicit sharing
- Connections and interaction
- Easy to reason about

Individual:

- Sequential is simple
- Concurrency applied carefully

A roadblock..

- Concurrent file I/O
- Sequential (other) system calls
- Inodes are shared (disk, memory)
- Standards (under) specification
- Need to 'lock' one or more inodes

Conclusions

- Compromise between sequential and highly concurrent
- Disciplined model of concurrency
- Iterative (and careful) introduction
- Techniques for addressing concurrency needs

A postlude on unit testing

Unit testing concurrency

Want to show that a slow/broken device doesn't break the system, unit testing the concurrency properties.

- Create a broken device
- Use channels to order operations
- Interfaces and concurrency

Setup

```go
type BlockDevice interface {
  Read(buf interface{}, pos int64) error
  Write(buf interface{}, pos int64) error
}

type BlockingDevice struct {
  BlockDevice
  HasBlocked chan int64 // position argument
  Unblock    chan bool
}

func (dev *BlockingDevice) Read(buf interface{}, pos int64) error {
  dev.HasBlocked <- pos
  <-dev.Unblock
  return dev.BlockDevice.Read(buf, pos)
}
```

Blocking a device

```
// create a block cache with test device, and a broken device
cache := createTestCache()
bdev := createBlockingDevice()
cache.MountDevice(1, bdev)

// to join the spawned goroutines (manual PAR)
done := make(chan bool)

go func() {
  // do a read on the broken device (1)
  cb := cache.GetBlock(1, SUPER_BLOCK)
  cache.PutBlock(cb)
  done <- true
}()
```

Testing the cache

```
go func() {
  // wait for the device to be blocked
  <-bdev.HasBlocked
  // then request a read from the non-broken device
  cb := cache.GetBlock(0, SUPER_BLOCK)

  // now release the broken device so it cleanly shuts down
  bdev.Unblock <- true
  cache.PutBlock(cb)
  done <- true
}()

// wait for both goroutines to finish
<-done
<-done
```

Questions? Comments?