15-440, Fall 2011, Class 05, Sept. 13, 2011
Randal E. Bryant

All code available in:
    /afs/cs.cmu.edu/academic/class/15440-f11/code/class05

Managing Concurrency

Useful references:
       http://golang.org/doc/GoCourseDay3.pdf

Coverage:
- Classical synchronization with locks & condition variables
- Using Go channels to control access to resources
- Using a client/server model to manage concurrent access to shared resources (style encouraged by Go).

Review. In 213, you learned the basics of concurrency.

Classical model:

Have set of threads running within an address space. Some parts of
state are shared, some are private. In typical application, establish
set of conventions:

* What data will be shared between threads
* How will we control access to shared data

Latter is done via various synchronization mechanisms. In particular,
you learned about semaphores:

Integer variable x that can operated on with two operations:

```
x.P():
        If x == 0, then block
        x--

x.V():
        x++
```

Both operations are done atomically, meaning that all steps take place
without any intervening operations.

Special case: Binary semaphore == Mutex:

x = 1: Unlocked. Resource is available
x = 0: Locked. Must wait to get to resource.

Common refer to operations P = "Lock" and V = "Unlock"

Let’s look at following problem:

Want to create FIFO queue that supports thread-safe operations:

```
b.Initialize()
        Initialize values

b.Insert(x)
        Insert item into queue

b.Remove().
        Block until queue not empty (if necessary)
        Return element at head of queue

b.Flush().
        Clear queue
```

Assume that we already have a sequential implementation of a buffer.
Suppose that b represented by structure with fields:

```
        sb: Sequential buffer implementation
        mutex: Mutual exclusion lock
```

Clearly need to wrap with mutex:

```
b.Initialize():
        b.sb = NewBuf()
        b.mutex = 1


b.Insert(x):
        b.mutex.lock()
        b.sb.Insert(x)
        b.mutex.unlock()

b.Remove():
        b.mutex.lock()
        x = b.sb.Remove()  # Oops. What if sb is empty?
        b.mutex.unlock()
        return x


b.Flush():
        b.mutex.lock()
        b.sb.Flush()
        b.mutex.unlock()
```

What’s wrong with this code?

Answer: If call Remove when buffer is empty, will call sb.Remove(),
which is invalid.

Let’s try to fix this.

Bryant & O’Hallaron, Figure 12.25 (p. 968) use semaphore "items" that
counts number of items in buffer

```
b.Initialize():
        b.sb = NewBuf()
        b.mutex = 1
        b.items = 0


b.Insert(x):
        b.lock()
        b.sb.Insert(x)
        b.mutex.unlock()
        b.items.V()

b.Remove():
        b.items.P()
        // This is the point of vulnerability.
        b.mutex.lock()
        x = b.sb.Remove()
        b.mutex.unlock()
        return x


b.Flush():
        b.mutex.lock()
        b.sb.Flush()
        b.items = 0
        b.mutex.unlock()
```

What’s wrong?

Answer: For just Insert & Remove, this would work fine. But Flush
messes things up. If flush occurs at point of vulnerability in
Remove, then again find self trying to remove from empty buffer.

Fixing race condition. How about this:

```
b.Remove():
        b.mutex.lock()
        b.items.P()
        // This is the point of vulnerability.
        x = b.sb.Remove()
        b.mutex.unlock()
        return x
```

Answer: Avoids race, but prone to DEADLOCK: reach point where no one is able to proceed.

In this case:

Remove when buffer is empty. Remove gets lock. Somewhere else, want to Insert, but can’t get past lock.

Find that it’s really hard to fix. My attempts with using binary semaphore to indicate whether or not buffer empty failed.

Better approach: Use CONDITION VARIABLES.

Condition variables provides synchronization point, where one thread
can suspend until activated by another.

Condition variable always associated with a mutex.
(Must have unique mutex for given cvar. One mutex can work with
multiple cvar’s).

Assume cvar connected to mutex:

```
cvar.Wait():
        Must be called after locking mutex.
        Atomically: release mutex & suspend operation

        When resume, lock mutex (but maybe not right away)

cvar.Signal():
        If no thread suspended, then no-op
        Wake up one suspended thread.
        (Typically do within scope of mutex, but not required)
```

Code for buffer with condition variables:

```
b.Initialize():
        b.sb = NewBuf()
        b.mutex = 1
        b.cvar = NewCond(b.mutex)


b.Insert(x):
        b.lock()
        b.sb.Insert(x)
        b.cvar.Signal()         # Optionally: Do only when previously empty
        b.mutex.unlock()

// First Version
b.Remove():
        b.mutex.lock()
        if b.sb.Empty() {
           b.cvar.Wait()   // Note that lock is first released & then retaken
        }
        x = b.sb.Remove()
        b.mutex.unlock()
        return x


b.Flush():
        b.mutex.lock()
        b.sb.Flush()
        b.mutex.unlock()
```

Remove isn’t quite right. Here’s the problem:

cvar.wait has 3 steps:

```
          Atomically { Release lock; suspend operation }

          ...

          Resume execution
          // Point of vulnerability
          Get lock
```

```
// Correct Version
b.Remove():
        b.mutex.lock()
        // Code looks weird.  But remember that are releasing and
        // regaining lock each time around loop.
        while b.sb.Empty() {
           b.cvar.Wait()   // Note that lock is first released & then retaken
        }
        x = b.sb.Remove()
        b.mutex.unlock()
        return x
```

Using Go channels

Go promotes a different view of concurrency, where set up miniature client/server structures within a single program. Use "channels" as mechanism for:

1. Passing information around
2. Synchronizing goroutines
3. Providing pointer to return location (like a "callback")

Basic idea:

```
Can make channel of any object type:
    * Bounded FIFO queue
      c := make(chan int, 17)
      d := make(chan string, 0)
    * Insertion: c <- 21
      If channel already full, then wait for receiver.
      Then put value at end
    * Removal    s := <- d
      If channel empty, then wait for sender
      Then get first value
```

Note that when channel has capacity 0, then insertion & removal are a "rendezvous"

Example: Use as mutex

```
type Mutex struct {
        mc chan int
}

// Create an unlocked mutex
func NewMutex() *Mutex {
        m := &Mutex{make(chan int, 1)}
        m.Unlock()                  # Initially, channel empty == locked
        return m
}

func (m *Mutex) Lock() {
        <- m.mc                      # Don’t care about value
}

func (m *Mutex) Unlock() {
        m.mc <- 1                    # Stick in value 1.
}
```

How about using channel to implement concurrent buffer:

* Acts as FIFO
* Allows concurrent insertion & removal

Shortcomings:

* Size bounded when initialized. Cannot implement bounded buffer

* No way to test for emptiness. When read from channel, cannot put back value at head position

* No way to flush

* No way to examine first element ("Front" operation)

Basic point:

* Channels are very low level.
* Most applications require building more structure on top of channels.

Method 1: Using channels for rendezvous:

Idea: Have goroutine for buffer that acts as traffic director:

* Receives request(s) on incoming channel(s)
* Selects one that may proceed
* Calling function does operation
* Tells director that it is done.

Find that need two request channels:

1. Operations that can proceed in any case
2. Operations that block if buffer is empty

```
    Read Ops   ---->|
    Other Ops  ---->|                  Director
                    |<---- Ack channel
```

When buffer empty, only accept requests from 1st channel.

Use Go operation "select" to choose between them when buffer nonempty.

Can share channel for Acking back to function.

Makes use of rendezvous property of channels

```
type Buf struct {
        sb *bufi.Buf               // Sequential buffer
        ackchan chan int           // Signals completion of operation
        readchan chan int          // Allows blocking when reading
        opchan chan int            // For nonblocking operations
}

func NewBuf() *Buf {
        bp := new(Buf)
        bp.sb = bufi.NewBuf()
        bp.ackchan = make(chan int)
        bp.readchan = make(chan int)
        bp.opchan = make(chan int)
        go bp.director()
        return bp
}

// Go routine to respond to requests
func (bp *Buf) director() {
        for {
                if bp.sb.Empty() {
                        // Enable only nonblocking operations
                        bp.opchan <- 1
                } else {
                        // Enable reads and other operations
                        select {     # Will allow only one communication
                        case bp.readchan <- 1:
                        case bp.opchan <- 1:
                        }
                }
                <- bp.ackchan // Wait until operations completed
        }
}

func (bp *Buf) startop() { <- bp.opchan }

func (bp *Buf) startread() { <- bp.readchan }

func (bp *Buf) finish() { bp.ackchan <- 1 }

func (bp *Buf) Insert(val interface{}) {
        bp.startop()
        bp.sb.Insert(val)
        bp.finish()
}

func (bp *Buf) Remove() interface{} {
        bp.startread()
        v := bp.sb.Remove()
        bp.finish()
        return v
}

func (bp *Buf) Empty() bool {
        bp.startop()
        rval := bp.sb.Empty()
        bp.finish()
        return rval
}

func (bp *Buf) Flush() {
        bp.startop()
        bp.sb.Flush()
        bp.finish()
}
```

```
Even More Go-Like:

Use channels to implement client/server model.

    Go routine that does all operations on buffer

    Functions supply requests into channel

    Request includes reply channel as "return address"

## This is how to get an enumerated type in Go
const (
        doinsert = iota
        doremove
        doflush
        doempty
)

## Message format.  Use same message format for all operation
## If operation does not require value, then use value nil.
## Reply will be either buffer value, nil, or boolean

type request struct {
        op   int                 // What operation is requested
        val  interface{}         // Optional value for operation
        replyc chan interface{} // Channel to which to send response
}

Version 1: Maintain two request channels:

type Buf struct {
        // Buffer has two request channels
        opc chan *request       // Nonblocking operations
        readc chan *request     // Operations that block for empty buffer
}

func NewBuf() *Buf {
        bp := &Buf{make(chan *request), make(chan *request)}
        go bp.runServer()
        return bp
}

func (bp *Buf) runServer () {
        // Create actual buffer
        sb := bufi.NewBuf()    // Note that this can be private to goroutine
        for  {
                var r *request
                if sb.Empty() {
                        r = <- bp.opc
                } else {
                        select {
                        case r1 := <- bp.opc:
                                r = r1
                        case r2 := <- bp.readc:
                                r = r2
                        }
                }
                switch r.op {
                case doinsert:
                        sb.Insert(r.val)
                        r.replyc <- nil
                case doremove:
                        v := sb.Remove()
                        r.replyc <- v
                case doflush:
                        sb.Flush()
                        r.replyc <- nil
                case doempty:
                        e := sb.Empty()
                        // Can send Boolean along channel
                        r.replyc <- e
                }
        }
}

func (bp *Buf) doop(op int, val interface{}) interface{} {
        r := &request{op, val, make(chan interface{})}
        bp.opc <- r
        v := <- r.replyc   ## Wait until operation completed
        return v
}

func (bp *Buf) doread(op int, val interface{}) interface{} {
        r := &request{op, val, make(chan interface{})}
        bp.readc <- r
        v := <- r.replyc   ## Wait until operation completed
        return v
}
```

Exported functions

```
func (bp *Buf) Insert(val interface{}) {
        bp.doop(doinsert, val)
}

func (bp *Buf) Remove() interface{} {
        return bp.doread(doremove, nil)
}

func (bp *Buf) Empty() bool {
        v := bp.doop(doempty, nil)
        e := v.(bool)
        return e
}

func (bp *Buf) Flush() {
        bp.doop(doflush, nil)
}
```

Final implementation. Same idea, but rather than using separate
channels, create buffer of "deferred" requests. We just happen to
have a suitable buffer data structure available!

```
// Which operations require waiting when buffer is empty
## This is the way to implement a set in Go.
var deferOnEmpty = map [int] bool { doremove : true }

## Same ideas as before
type request struct {
        op   int                // What operation is requested
        val  interface{}        // Optional value for operation
        replyc chan interface{} // Channel to which to send response
}

## Only one channel to implement external interface
type Buf struct {
        requestc chan *request  // Request channel for buffer
}

func NewBuf() *Buf {
        bp := &Buf{make(chan *request)}
        go bp.runServer()
        return bp
}

func (bp *Buf) runServer () {
        // Buffer to hold data
        sb := bufi.NewBuf()
        // Buffer to hold deferred requests
        db := bufi.NewBuf()
        for  {
                var r *request
                ## No need for select.  We do our own scheduling!
                if !sb.Empty() && !db.Empty() {
                        // Revisit deferred operation
                        r = db.Remove().(*request)
                } else {
                        r = <- bp.requestc
                        if sb.Empty() && deferOnEmpty[r.op] {
                                // Must defer this operation
                                db.Insert(r)
                                continue
                        }
                }
                switch r.op {
                case doinsert:
                        sb.Insert(r.val)
                        r.replyc <- nil
                case doremove:
                        v := sb.Remove()
                        r.replyc <- v
                case doflush:
                        sb.Flush()
                        r.replyc <- nil
                case doempty:
                        e := sb.Empty()
                        // Can send Boolean along channel
                        r.replyc <- e
                case dofront:
                        v := sb.Front()
                        r.replyc <- v
                }
        }
}
```

```
func (bp *Buf) dorequest(op int, val interface{}) interface{} {
        r := &request{op, val, make(chan interface{})}
        bp.requestc <- r
        v := <- r.replyc
        return v
}


func (bp *Buf) Insert(val interface{}) {
        bp.dorequest(doinsert, val)
}

func (bp *Buf) Remove() interface{} {
        return bp.dorequest(doremove, nil)
}

func (bp *Buf) Empty() bool {
        v := bp.dorequest(doempty, nil)
        e := v.(bool)
        return e
}

func (bp *Buf) Flush() {
        bp.dorequest(doflush, nil)
}
```