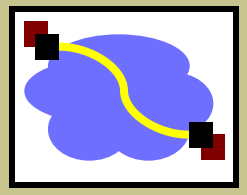
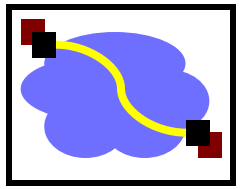


15-440 Distributed Systems

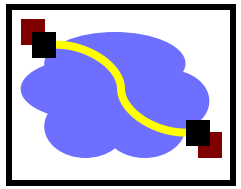
Lecture 04 – Classical Synchronization
GO Style Concurrency
Thursday, September 6th, 2018

Logistics Updates



- P0 Released (Tuesday, Sept 4th)
  - Due: (Sept 13th, 11:59pm EST)
  - Recitation was on Wednesday, Sept 5th
- HW1 released next week (Sept 12)
  - Due Sept 9/23 (*No Late Days*).
- As always, check for due date on the web / writeup
- Waitlists: everyone is enrolled, no more additions.

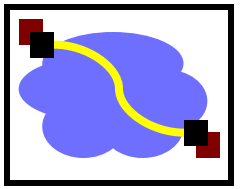
Today's Lecture Outline



- Concurrency Management - Synchronization
- Part I: Review of Classical Concurrency
  - Concepts, Locks, Condition Variables,
- Part II: Concurrency Model in GO
  - Quick review of yesterday's recitation on GO Overview

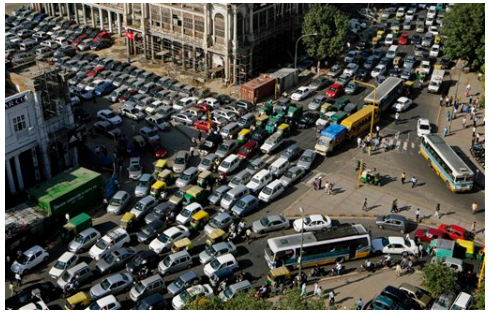
    For more details, look at the slides for recitation

Concurrency



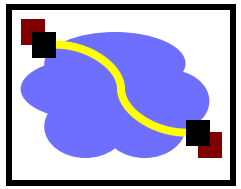
- We will use concurrency concepts repeatedly
- Recap: Why is concurrency important/useful?
  - Allows safe/multiplexed access to shared resources
  - i.e. safe operation with multiple independent entities
    - Single core CPUs to wide area distributed systems
  - Why is sharing resources useful? Real life example?

With (some at least!)
Concurrency Control



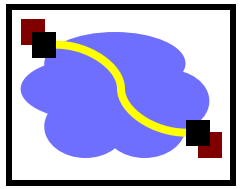
…or not!

Concurrency is key in DS



- Today, we start with threads on a single node …. we will expand them to multiple machines
  - Key assumption: ignore independent failures
- Code Concurrency: Terms (Ref: Dijkstra ‘65,’68)
  - **Critical Section**: piece of code accessing a shared resource, usually variables or data structures
  - **Race Condition**: Multiple threads of execution enter CS at the same time, update shared resource, leading to undesirable outcome
  - **Indeterminate Program**: One or more Race Conditions, output of program depending on ordering, non deterministic
  - So, what is the solution?
    - **Mutual Exclusion!**

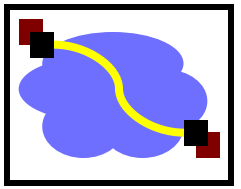
Achieving Mutual Exclusion



- **Mutual Exclusion**: guarantee that only a single thread/process enters a CS, avoiding races
- Desired Properties of Mutual Exclusion
  - **Mutual Exclusion (Correctness)**: single process in CS at one time
  - **Progress (Efficiency):** Processes don't wait for available resources, or no spin-locks => no wasted resources
  - **Bounded Waiting (Fairness)**: No process waits for ever for a resource, i.e. a notion of fairness

  - Trivial solution if we didn't care about fairness! What is it?
  - Can you give an example of concurrent access to files in an OS which does not lead to concurrency problems or need mutex?

Achieving Mutual Exclusion



- Mutual Exclusion: Usually need some H/W support
  - Test-and-Set instruction (**atomic** testing/setting of a value)

```
TS (<memloc>)
{  if <memloc>==1
     {
         <memloc>=0;
         return 1;
     }
   else
         return 0;
}

Note: Atomic/Non-interruptable

We can use Test-and-Set to
implement Mutex
```

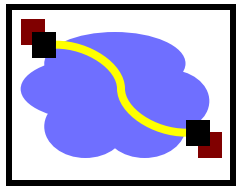
```
// Do this before entering CS
Acquire_Mutex(<mutex>)
 {
      while(!TS(<mutex>))
 }
```

```
{CS: Critical Section of Code}
```

```
// After Exiting CS
Release_Mutex(<mutex>)
 {
      <mutex> = 1
 }
```

- Build more complex primitives around this H/W

Classical Model of Concurrency

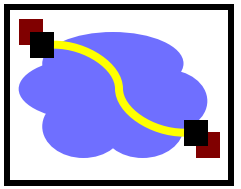


- Threads running within an address space
  - Private/Shared state => primitive to access shared state
  - E.g. Semaphores: Integer variable ‘x’ with 2 operations

```
x.P():
     while (x == 0) wait;
     x–

x.V():
     x++

Note: `P’ and `V’ operations are
Atomic, not interruptable

What is a Binary Semaphore then?
```

```
// binary Semaphore = Mutex
x=1; Unlocked, Resource Available
x=0; Locked, Must wait for resource
```

Semaphones to Create a FIFO



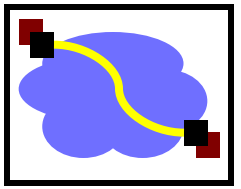
- Idea: Want to create (thread safe) FIFO queue
  - Similar to GO channel, without a capacity limit

```
b.Init():
    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 we have a sequential implementation of a FIFO buffer
// b represents a structure with fields
    sb: Sequential buffer implementation
    mutex: Mutual exclusion lock
```

- What do we need to do next? Wrap with Mutex

Thread Safe FIFO Queue



```
b.Init():
    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()
    b.mutex.unlock()
    return x

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

Is this correct?

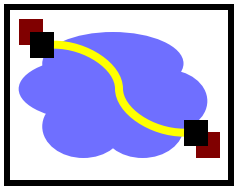
No, what if Remove is called and the buffer is empty?

```
.
.
.
.
b.Remove():
  retry:
    b.mutex.lock()
    if !(b.sb.len() > 0) {
       b.mutex.unlock()
       goto retry
    }
.
.
.
```

Potential Fix. Does this do it?

No, not really. This is a spin-lock. Wastes resources and unclear if a Insert(x) will ever make progress.
This is inefficient and can be a potential LIVELOCK.

Thread Safe FIFO Queue



- Use semaphore “items” (Bryant and O’hallaron)

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

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

b.Remove():
    b.items.P()
    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()
```

This fixes it.

No, what if Flush is called after b.items.P() and the buffer is empty?

```
.
.
.
b.Remove():
    b.mutex.lock()
    b.items.P()
    x = b.sb.Remove()
    b.mutex.unlock()
    return x
.
.
.
```

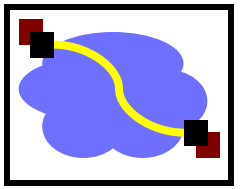
OK, now we are good!

No, not really. We avoid the race condition but prone to a **DEADLOCK**. Can reach point where no one is able to proceed. How?

Lets say you call Remove when buffer is empty. Remove gets lock. Somewhere else, want to Insert, but can't get past lock.

Hard to fix, need different approach.

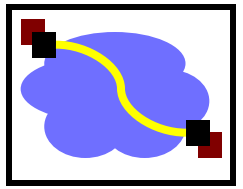
Buffer using ConditionVars



- Lets use Condition Variables (cvars)
  - cvars provide a sync point, one thread suspended until activated by another. (more efficient way to wait than spin lock)
  - cvar always associated with mutex
  - Wait() and Signal() operations defined with cvars

```
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 (at least) one suspended thread.
    (Typically do within scope of mutex, but not required)
```

Buffer using ConditionVars

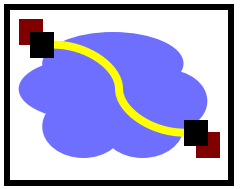


```
b.Init():
    b.sb = NewBuf()
    b.mutex = 1
    b.cvar = NewCond(b.mutex)

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

b.Remove():
    b.mutex.lock()
    if b.sb.Empty() {
        b.cvar.wait()
    }
    x = b.sb.Remove()
    b.mutex.unlock()
    return x

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

```
//  Signal() If no thread suspended, then NO-OP
    Wake up (at least) one suspended thread.
    (Typically do within scope of mutex, but not required)
```

```
// wait() Note that lock is first released & then retaken
   Atomically: release mutex & suspend operation
   When resume, lock mutex (but maybe not right away)
```

Buffer using ConditionVars



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

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

b.Remove():
    b.mutex.lock()
    if b.sb.Empty() {
        b.cvar.wait()
    }
    x = b.sb.Remove()
    b.mutex.unlock()
    return x

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

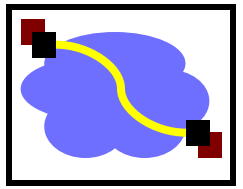
Still a small problem.

```
Cvar.wait()  // 3 steps

Atomically {release lock + suspend operation)
.
.
.
Resume Execution
.
// point of vulnerability, someone can flush here
.
Lock Mutex
```

What a signal means:

Mesa semantics (looser) vs Hoare Semantics (tighter)

Buffer using ConditionVars



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

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

b.Remove():
    b.mutex.lock()
    while b.sb.Empty() {
        b.cvar.wait()
    }
    x = b.sb.Remove()
    b.mutex.unlock()
    return x

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

Change “if” to While

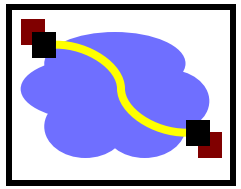
```
// What happens
    Lock

    if !sb.empty() goto ready
    Unlock
    wait for signal
    Lock

    if !sb.empty() goto ready
    Unlock
    wait for signal
    Lock
    . . .
ready: Can safely assume have lock & that buffer nonempty
```

```
// wait() Note that lock is first released & then retaken
   Atomically: release mutex & suspend operation
   When resume, lock mutex (but maybe not right away)
```

Simple Rule: With Mesa semantics, use while loops to recheck the condition. Always safe to do so.

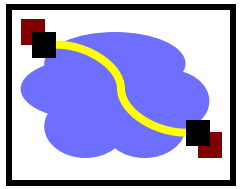
Additional examples for Cvars in the Remzi Ref.

Today's Lecture Outline



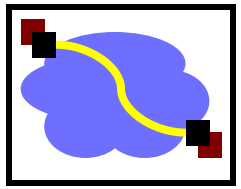
- Concurrency Management - Synchronization
- Part I: Review of Classical Concurrency
  - Concepts, Locks, Condition Variables,
- Part II: Concurrency Model in GO
  - Quick review of yesterday's recitation on GO Overview
    For details, look at the slides for recitation

Concurrency model for GO



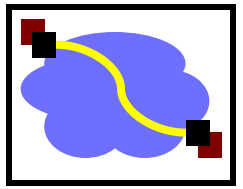
- Set up a mini Client/Server within programs
  - “Channels” and “GOroutines”
- Channels are used for:
  - Passing information around (are typed: int, char, …)
  - Synchronizing GOroutines
  - Providing pointer to return location (like a "callback")
- GoRoutines:
  - Independently executing function, launched by “go”
  - Independent call stack, very inexpensive, 1000s of them
- Concept: Instead of communicating by sharing memory, share memory by communicating

Concurrency vs Parellelism



- Source: GO concurrency Bob Pike (Google)
- Concurrency is not parallelism, although it enables parallelism
- 1 Processor: Program can still be concurrent but not parallel
- However a well written concurrent program may run well on multiprocessor platform

GO Concurrency



- Make Channel of any object Type Source
  - Bounded FIFO queue

```
c := make(chan int, 17)
d := make(chan string, 0)
```

  - Insertion (If channel full, wait for receiver). Then put value at the end.

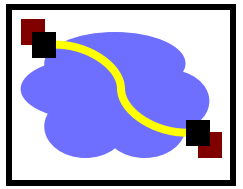
```
c <- 21
```

  - Removal (If channel empty, then wait for sender. Then get first value)

```
s := <- d
```

  - Note, when channel capacity is 0, Insert/Remove is a rendezvous. i.e. sync point

GO Concurrency – Examples



- Patterns
  - Capacity = 0; Sync Send/Rcv

```
insert          remove
------>    |  ------->
```

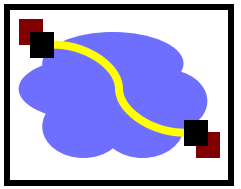
  - Capacity = 1: Pass token from sender to receiver

```
insert          remove
------>   |x|  ------->
```

  - Capacity = n: Bounded FIFO (e.g. n=5)

```
insert          remove
------>   |x x x x x|  ------->
```

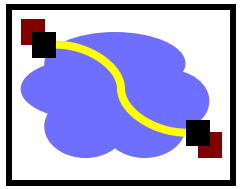
GO Concurrency – Mutex



- Use GO Channels to implement a 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
}

// Lock operation, take a value from the channel
func (m *Mutex) Lock() {
     <- m.mc # Don't care about value
}

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

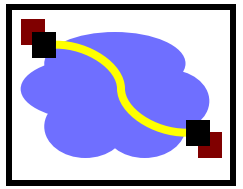
GO Concurrency – Limitations



- GO Channels have some limitations
  - Size: Bounded when initialized, cannot have unbounded buffer
  - No way to test for emptiness. When read from channel cannot put back value to head of channel.
  - No way to flush channel
  - No way to examine first element

- Point 1: GO channels low level primitives. Most Apps will need you to build more structure on top
- Point 2: GO also has support for mutex/cvars/etc

Summary



- Concurrency is a style of programming
  - Different scales, from single node to large DSs
  - Requires Mutual Exclusion: Mutex, Efficiency, Fairness
- Various Primitives:
  - Semaphores, Mutexes, Condition Variables
  - Hard to write concurrent programs that are correct under all conditions
- GO Concurrency Model
  - Channels and GoRoutines
  - Use channels for communication, instead of Mutex/Sem
  - Different concurrency patterns, several limitations

Questions?


Wednesday January 08, 2003
HOW DO I MAKE THIS SOFTWARE SCHEDULE ONE PERSON TO TWO TASKS AT THE SAME TIME?
I CAN WRITE A PATCH THAT INSERTS NEW MONTHS IN THE TIMELINE.
AND THE SECOND TASK IS DUE ON THE FIFTEENTH OF FLOOPUARY.
www.dilbert.com scottadams@aol.com
© 2002 United Feature Syndicate, Inc.

Backup