Concurrent Programming

COMP 524: Programming Languages
Srinivas Krishnan
April 21, 2011

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

Based in part on slides and notes by Vitaly Shmatikov (UT-Austin) and Google Go Team
Concurrency

Northeast Blackout of 2003
Affected 50 million people in U.S. and Canada

Race condition in alarm management system caused it to stall, alarms backed up and stalled both primary and backup server

“We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it.”
-- GE Energy's Mike Unum
Remember This?

The Evolution Of Computer Programming Languages

Concurrent Pascal
Introduction

Concurrent programming is hard

- Traditional OS model has for the longest time supported “Multiprogramming”
- Multiple Processes running on a single CPU interleaved.
- CPU scheduler responsible for context-switching and selecting another process to run
Enter Era of Multicore

Need to run multiple events at the same time on separate processors

➡ Requires a new paradigm shift
➡ Management of resources and scheduling
➡ New Issues: Deadlocks, Race Conditions
➡ New and more complicated Bugs !!!!

![Diagram showing processes and CPUs]
Benefits

Speed

→ If a task takes time ‘t’ on one processor, on ‘n’ processors should the time be t/n

Availability

→ Spawn a new process if one is busy and have it immediately execute on a new core.

Distribution

→ Multiple processors in different locations can work on different stages of a problem

We do it all the time, why can’t we program computer to do the same?

→ We talk and hear at the same time!!
Examples

Webserver

- Receive a request, process the request, send back a reply
- Embarrassingly parallel

![Diagram showing the process of request, process, and reply]
Examples

Webserver

- Receive a request, process the request, send back a reply
- Embarrassingly parallel
Examples

Webserver

- Receive a request, process the request, send back a reply
- Embarrassingly parallel
Examples

Webserver

- Receive a request, process the request, send back a reply
- Embarrassingly parallel

Diagram:
- Multiple requests
- Process
- Send back replies
Challenges

- Harder to write concurrent code correctly
  - “need at least an order of magnitude in speedup for concurrent program to be worth the effort”
- Lot of problems are inherently sequential
  - Packet processing sequential, need to go in time order
  - Circuit evaluation
  - In Practice: Usually multiple stages need to coordinate various pieces.
- Specific
  - Communication: Waiting on messages to be sent or received
  - Synchronization: Waiting on another process
  - Atomicity: Cannot leave within a section of code
Language Support

Threads

- Think of a thread as a system “object” containing the state of execution of a sequence of function calls
- Each thread needs a separate run-time stack (why?)
- Pass threads as arguments, return as function results

Communication

- Synchronous communication
- Asynchronous buffers that preserve message order

Concurrency

- Locking and mutual exclusion
- Atomicity is more abstract, less commonly provided
Explicit vs Implicit Concurrency

Explicit

- Fork or create threads/processes explicitly
- Explicit communication between processes

Implicit

- Rely on compiler to identify potential parallelism
- Instruction-level and loop-level parallelism can be inferred, but inferring subroutine-level parallelism has had less success
- Java ArrayList use implicit concurrency

```
List<Item> list = ...
while (Item item : list) {
  if (isOld(item)) {
    list.remove(item);
  }
}
```

```
List<Item> list = ...
Iterator<Item> iterator = list.iterator();
while (iterator.hasNext()) {
  Item item = iterator.next();
  if (isOld(item)) {
    iterator.remove();
  }
}
```
Java Threads

Thread
- Set of instructions to be executed one at a time, in a specified order
- Special Thread class is part of the core language. C/C++ its a add-on e.g. PThreads, Intel TBB

Methods of Class Thread
- start : method called to spawn a new thread
  - Causes JVM to call run() method on object
- suspend : freeze execution
- interrupt : freeze and throw exception to thread
- stop : forcibly cause thread to halt
public class Thread implements Runnable {
    private char name[];
    private Runnable target;
    ...
    public final static int MIN_PRIORITY = 1;
    public final static int NORM_PRIORITY = 5;
    public final static int MAX_PRIORITY = 10;

    private void init(ThreadGroup g, Runnable target, String name) {...}

    public Thread() { init(null, null, "Thread-" + nextThreadNum()); }
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum());
    }
    public Thread(Runnable target, String name) { init(null, target, name); }

    public synchronized native void start();

    public void run() {
        if (target != null)
            target.run();
    }
}
States of a Java Thread

- **Non-existing**
  - create thread object

- **New**
  - start
  - destroy

- **Runnable**
  - wait, join
  - notify, notifyAll
  - thread termination
  - destroy
  - run method exits

- **Blocked**
  - wait, join
  - notify, notifyAll
  - thread termination
  - destroy

- **Terminated (Dead)**
  - garbage collected and finalization
  - destroy

- **Non-Existing**
  - destroy

Thread may not execute any “finally” clauses, leave objects locked.
Concurrent Garbage Collection

♦ Need to stop thread while mark-and-sweeping
  • Do other threads need to be stopped?

♦ Problem: objects may change during collection

♦ Solution: prevent read/write to memory area
  • Subtle!
    • Generational GC distinguishes short-lived and long-lived objects
    • Copying collectors allows reads from old area if writes are blocked…
POSIX Threads

- Pthreads library for C

- **pthread_create** - create a new thread giving it a “starting” procedure to run along with a single argument.
- **pthread_self** - ask the currently running thread for its thread id.
- **pthread_join** - join with a thread using its thread id (an integer value)

- **pthread_mutex_init** - initialize a mutex structure
- **pthread_mutex_destroy** - destroy a mutex structure
- **pthread_mutex_lock** - lock an initialized mutex, if already locked suspend execution and wait
- **pthread_mutex_trylock** - try to lock a mutex and if unsuccessful, do not suspend execution
- **pthread_mutex_unlock** - unlock a mutex that was locked by the current thread

- **pthread_cond_init** - initialize a condition variable structure
- **pthread_cond_destroy** - destroy a condition variable structure
- **pthread_cond_wait** - block the currently running thread on a condition variable indefinitely
- **pthread_cond_timedwait** - block the currently running thread on a condition variable for a specific time
- **pthread_cond_signal** - wake up one thread blocked on a condition variable
- **pthread_cond_broadcast** - wake up all threads blocked on a condition variable
Example of Using POSIX Threads

```c
#include <pthread.h>
#include <unistd.h>  /* sleep declaration */
#include <stdio.h>   /* printf declaration */
const int NUM_THREADS = 5;

void* sleeping(void* st)
{
    int sleep_time = (int) st; /* cast void* to an int */
    printf (“thread %d sleeping %d seconds ...
”, pthread_self(), sleep_time);
    sleep(sleep_time);
    printf (”\nthread %d awakening
”, pthread_self());
}

main( int argc, char *argv[] )
{
    pthread_t tid[NUM_THREADS]; /* array of thread IDs */
    int i;

    for ( i = 0; i < NUM_THREADS; i++ )
        pthread_create (&tid[i], NULL, sleeping, i+2);

    for ( i = 0; i < NUM_THREADS; i++ )
        pthread_join (tid[i], NULL);

    printf (“main() reporting that all %d threads have terminated\n”, i);
} /* main */
```
Thread Stacks

Multiple thread run-time stacks, each a separate “thread of execution”

Main thread and run-time stack

```c
main() {
  for(i=0; i < n; i++)
    pthread_create(...)
  ...
}
```

Stack Activation Frames

1MB Thread Stack

- Underflow stack frame raises exception if “popped”
- Starting procedure activation frame
- Nested procedure call activation frames

`red-lined` overflow page to generate memory segmentation violation (SEGv) if an overflow occurs from trying to create a frame beyond the end of the stack
Here Come’s Go !!!

 ➪ Developed by Robert Griesemer, Ken Thompson, and Rob Pike since late 2007.

 ➪ Recent Developments influence the design
   ‣ Multicore Systems
   ‣ Google’s Map-Reduce Model
   ‣ Async Computation: Rise of Networking

 ➪ Lowering Compile time:
   ‣ It takes too long to build software.
   ‣ Dependencies are large
As xkcd observes...

http://xkcd.com/303/

http://xkcd.com/303/
Strong Typing

Is a pain !!!

➡ Robert Griesemer: “Clumsy type systems drive people to dynamically typed languages.”

➡ Clunky typing: Taints good idea with bad implementation. Makes programming harder (think of C's `const`: well-intentioned but awkward in practice).
Stated Goals

- The efficiency of a statically-typed compiled language with the ease of programming of a dynamic language
- Safety: type-safe and memory-safe.
- Good support for concurrency and communication.
- Efficient, latency-free garbage collection.
- High-speed compilation.
Design

“Go is a general-purpose language designed with systems programming in mind”

- Simple regular grammar
  - No Symbol Table needed (why?)
- Reduce typing: No needless typing
  - `foo.Foo *myFoo = new foo.Foo(foo.FOO_INIT)`
- Type System:
  - Lightweight: Dynamic Typing
  - No implicit conversions: keep things explicit.
Design

Runtime System:
- Garbage collection.
- Strings, maps, communication channels.
- Implicit Concurrency.

Package model:
- Explicit dependencies to enable faster builds.
Concurrency in Go

Garage Collected Processes

- "goroutines" built using language support
- Needs runtime to work

Management

- At runtime memory management
- Stack growing is handled by the RTS
- Scheduler manages multiplexing of "goroutines"
Types

Constants:
- `const N = 1024`

Variables
- `var x, y, *float`
- `var v1 int`
- `var v2 []int`

Pointers
- `var p = &v2 // Type inferred to be a pointer`
Types

Composite Data Types

➡ Structs

type T struct {
    name string // name of the object
    value int   // its value
}
var v9 myStruct              // v9 has structure type
var p9 *myStruct             // p9 is a pointer to a structure

➡ Strings

type String struct {
    data
    length
}

Interfaces

Python Inspired “Duck Typing”

➡Pure Abstract class, simply specifies the behavior

type myInterface interface {
    get() int
    set(i int)
}
func (p *myType) set(i int) { p.i = i }

func getAndSet(x myInterface) {} 
func f1() {
    var p myType 
    getAndSet(&p)
}
goroutines

“Do not communicate by sharing memory; instead, share memory by communicating.”

- Shared Data values are passed never accessed by different threads
- Race conditions cannot happen by design

**Definition**

- **Functions** executing in parallel in the same address space
- **Cost:** Allocate Stack Space
- Stacks dynamically grow and managed by the RTS
- No need to worry about how to create or manage threads
- Each goroutine multiplexed onto OS threads
goroutines

Prefix a function by keyword “go”

go list.Sort() // run list.Sort in parallel; don't wait for it.

    func server(i int) {
        for {
            print(i)
            sys.sleep(10)
        }
    }

    go server(1)
    go server(2)
goroutine Channels

Used to communicate across goroutines

Send using `<-` as a binary op, Receive using `<-` as a unary op

type cmd struct { get bool; val int }
func manager(ch chan cmd) {
    var val int = 0
    for {
        c := <- ch // Receive
        if c.get { c.val = val; ch <- c } // Send
        else { val = c.val }
    }
}
**Parallelize**

**Force parallelization across cores**

```go
// Expensive Vector ops
type Vector []float64

// Apply the operation to v[i], v[i+1] ... up to v[n-1].
func (v Vector) DoSome(i, n int, u Vector, c chan int) {
    for ; i < n; i++ {
        v[i] += u.Op(v[i])
    }
    c <- 1 // signal that this piece is done
}

// Launch above code across 4 cores
const NCPU = 4 // number of CPU cores

func (v Vector) DoAll(u Vector) {
    c := make(chan int, NCPU) // Buffering optional but sensible.
    for i := 0; i < NCPU; i++ {
        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
    }
    // Drain the channel.
    for i := 0; i < NCPU; i++ {
        <-c // wait for one task to complete
    }
    // All done.
}
```
func (server *serverType) accept(lis net.Listener)
{
    for {
        conn, err := lis.Accept()
        if err != nil {
        }
        go ServeConn(conn)
    }
}

func (server *serverType) process(conn)
{
    arith := new(Arith)
    rpc.Register(arith)
    rpc.HandleHTTP()
    l, e := net.Listen("tcp", ":1234")
    if e != nil {
        log.Exit("listen error:", e)
    }
    go http.Serve(conn, Arith)
}
func serveHTTP(c *http.Conn, req *http.Request) {
    if req.Method != "CONNECT" {
        c.SetHeader("Content-Type", "text/plain; charset=utf-8")
        c.WriteHeader(http.StatusMethodNotAllowed)
        io.WriteString(c, "405 must CONNECT to "+rpcPath+"\n")
        return
    }
    conn, _, err := c.Hijack()
    if err != nil {
        return
    }
    io.WriteString(conn, "HTTP/1.0 "+connected+"\n\n")
    ServeConn(conn)
}