CS 839: Systems Verification

Lecture 17: Concurrency intro

Learning outcomes

1. Formalize what a concurrent program does
2. Understand why reasoning about concurrency is challenging

Why Concurrency?

Hardware has many cores

Systems software must be concurrent

Challenge: many interleavings

- Even short code has exponentially many executions
- Testing every interleaving isn't feasible

Modern CPUs have many cores, and concurrency is important to get good performance from the machine. Systems software often needs to be concurrent so that applications can get good performance. The fundamental challenge is the number of interleavings: a small amount of code can now execute in many ways. How do we make sure every interleaving works correctly? Testing every interleaving isn't feasible, which makes verification more attractive in this setting.

Concurrency in the real world

- **Linux processes:** private memory, shared filesystem/network
- **Threads:** private stacks, shared memory
- **Go goroutines**
- **Web Workers:** share with message passing
- **Virtual machines:** isolation

When we program a computer, we can access concurrency in a number of ways. Linux has processes that run concurrently with their own memory. A process can have multiple threads that share memory but have distinct stacks. Go has goroutines managed by the runtime. Browsers have Web Workers. Hypervisors can run multiple VMs. In each case, there are multiple units of execution with different spawning mechanisms and different shared state, but similar issues arise when threads interact with shared state.

Terminology

Thread: Independent unit of execution

Parallelism: Simultaneous execution on different cores

Modeling concurrency: Atomic actions that interleave

We use "thread" generically for an independent unit of execution. Parallelism sometimes means threads running truly simultaneously on different CPU cores. When we consider threads to run "simultaneously" it's sufficient to consider each thread as having atomic actions (imagine individual CPU instructions) which interleave in some single global order. When two atomic actions happen simultaneously on separate cores, this produces two interleavings, one for each order.

In Go, the `go` statement makes it easy to run something concurrently. The syntax is `go f()` where the argument must be a function call. This runs `f()` in another goroutine - a separate thread of execution. The spawned goroutine gets its own stack but otherwise all goroutines share access to the same variables. In the example, we could see either output order, or if this is in `main`, the background goroutine might not finish at all.

Exercise: what else could happen?

```
go func() {
  fmt.Println("hello 1")
}()
fmt.Println("hello 2")
```

answer: could interleave within prints, "hehello 1\nllo 2\n"

Could expect (a) Println is always one `write(2)` system call, and (b) `write(2)` on stdout is atomic

but these aren't guarantees and anyway do depend on the size of the buffer

Modeling concurrency with a threadpool

New primitive: $\mathrm{spawn}\ e$

Previously: $(e, h) \rightsquigarrow (e', h')$

Now: $(T, h) \rightsquigarrow_{tp} (T', h')$

We add a new language construct $\backslash\mathrm{spawn}\ e$ which creates a new thread running e and evaluates to the unit value. Previously we had semantics for heap programs $(e, h) \rightsquigarrow (e', h')$. Now we're no longer running one expression e but a whole list called a **thread pool** T. The semantics has the form $(T, h) \rightsquigarrow_{tp} (T', h')$. We have a single global heap and the only state within one thread is the expression itself - no thread metadata or thread-local storage.

Thread Pool Semantics

$$\frac{(e, h) \rightsquigarrow (e', h')}{(T_1 +\!\!+ [e] +\!\!+ T_2, h) \rightsquigarrow_{tp} (T_1 +\!\!+ [e'] +\!\!+ T_2, h')}$$

Pick a thread, execute it one step, possibly switch to another

The threadpool semantics is closely related to the semantics of individual expressions. For the most part, transitions pick some thread, execute it for a bit, and then possibly switch to another thread. If a threadpool is running some thread e in the middle, it has the shape $T_1 + [e] + T_2$. The transition changes only e to e' and the heap from h to h'. This models the thread e being chosen to run next - the semantics only ever runs one thread at a time.

The Spawn Rule

$$(T_1 +\!\!+ [K[\text{spawn}\, e]] +\!\!+ T_2, h) \rightsquigarrow_{tp} (T_1 +\!\!+ [K[()]] +\!\!+ T_2 +\!\!+ [e], h)$$

Heap uninvolved
Creates new thread at end of pool
Returns $()$ to spawning thread, within an evaluation context

The spawn construct is the one thing that expands the threadpool. The heap is uninvolved, and the only effect is to create a new thread. Recall that $K[\text{spawn}\, e]$ means spawn is the next thing to run. The semantics say to launch a new thread, then continue running the rest of the program with $\text{spawn}\, e$ having returned $()$.

Soundness for concurrent programs

Previously: if P, e doesn't get stuck, and if it terminates in v, Q(v)

Now: if P,

- none of the spawned threads get stuck
- if "main thread" terminates in v, Q(v)
- background threads can terminate in any value

Exercise: is this the right model?

Consider two contexts:

- goroutines in the Go runtime
- processes in Linux

Concurrency bugs: race conditions

```
var counter uint64
go func() {
  counter = counter + 1
}()
go func() {
  counter = counter + 1
}()
```

Say counter is at `0x80a1c`. Both threads run this code:

```
mov 0x80a1c, %eax
add $0x1, %eax
mov %eax, 0x80a1c
```

Good Interleaving

```
# counter = 10
mov 0x80a1c, %eax
add $0x1, %eax
mov %eax, 0x80a1c
                        mov 0x80a1c, %eax
                        add $0x1, %eax
                        mov %eax, 0x80a1c
# counter = 12
```

Here's one interleaving where everything works correctly. The first thread completes all three instructions, then the second thread runs. We start with counter at 10 and end up with 12, as expected.

Bad Interleaving

```
# counter = 10
mov 0x80a1c, %eax
                        mov 0x80a1c, %eax
                        add $0x1, %eax
add $0x1, %eax
                        mov %eax, 0x80a1c
mov %eax, 0x80a1c
# counter = 11 (bug!)
```

Here's a problematic interleaving. Both threads load the value 10, both add 1 to get 11, and both store 11. The final result is 11 instead of 12. What went wrong? The two accesses to `counter` form a race condition: two simultaneous operations on the same memory, where at least one is a write. The problem is that `counter = counter + 1` didn't run atomically as desired.

Puzzle: Loop Increment

```
var counter = 0
go func() {
  for i := 0; i < 10; i++ {
    counter = counter + 1
  }
}()
go func() {
  for i := 0; i < 10; i++ {
    counter = counter + 1
  }
}()
```

Questions:

- Maximum final value? (hint: it's 20)
- **Minimum final value?**

Here's a puzzle. Consider incrementing in a loop instead. As before, each increment is three separate steps. The maximum value is obviously 20. But what is the minimum value? This is harder than it looks - think about it before continuing.

Puzzle Solution

Answer: 2

Observation: addition and storing can be viewed as atomic

trace showing an outcome of 2

thread 1	thread 2
x := !ctr (0)	x := !ctr (0)
	... (9 iterations)
ctr ← x + 1 (1)	
	x := !ctr (1)
... (9 iterations)	
ctr ← x + 1 (?)	
(done)	
	ctr ← x + 1 (2)
	(done)

The answer is 2. We can think of the loop body as two steps: load and then increment-and-store. Here's the key trace: Thread 1 loads 0, then Thread 2 loads 0 and completes 9 iterations. Thread 1 stores 1. Thread 2 loads 1. Thread 1 completes its remaining 9 iterations. Thread 2 finally stores 2. This demonstrates how badly things can go wrong with race conditions.

Using concurrency safely

Synchronization primitives control interleavings

This lecture: mutexes and a barrier (WaitGroup)

Mutexes: Mutual Exclusion

```
var m sync.Mutex
go func() {
  m.Lock()
  fmt.Println("hello...")
  fmt.Println("world")
  m.Unlock()
}()
m.Lock()
fmt.Println("bye...")
fmt.Println("there")
m.Unlock()
```

Only one thread can be inside a critical section at a time.

Mutexes or locks provide mutual exclusion. If threads call `m.Lock()` and `m.Unlock()` around a critical section, only one thread can be inside at any time. In this example, there's no doubt about interleavings: each critical section must run atomically, so even with multiple print statements we expect only two possible orderings.

WaitGroup: Barriers

```
func printTwo() {
  var wg sync.WaitGroup
  wg.Add(2)
  go func() {
    fmt.Println("hello 1")
    wg.Done()
  }()
  go func() {
    fmt.Println("hello 2")
    wg.Done()
  }()
  wg.Wait()
}
```

Always prints both messages (in either order), then returns.

Go's `sync.WaitGroup` is a form of barrier used to wait for threads to finish. This always prints "hello 1" and "hello 2" in either order, and then returns. If we call this in `main`, both print statements are guaranteed to run. Note this is not the same feature as locks - there's no mutual exclusion here. The fundamental primitive is waiting for something to happen.