Lecture 2

2020年4月24日 星期五　　上午3:30

Thread is the most important tool to manage concurrency.

Each thread has its own counter, register and stack.

Reasons of using threads:

1. I/O concurrency
2. Parallelism
3. Convenient reason: periodically check something

What can we do if we don't use thread?

Ans: Event-driven programming (asychronous programming)
It has a single thread and a single loop, what the loop does is sitting there and waiting for any input or request.
Problem: once reach I/O concurrency, it can't get CPU parallelism

Process vs. Thread

- Process: is a single program that you're running, has a single address space and a single bunch of memory for the process.
- Threads: inside a process, you might have multiple threads.

Example: When you're running a Go program, running the Go program creates one UNIX process and one sort of memory area and when the Go process creates Go routines, they are sitting inside the process.

No interactions between process, but multiple threads can share memory, mutex of the same process.

Thread challenges:

- Data sharing among all threads

Example: a global variable N, always incrementing itself
N = N+1
when a thread is executing the code, the other thread may also executing this code. The result will be incorrect
It's called "race" (*#)
Solution: add a lock to the data
{ mu.lock()
n = n+1
mu.unlock()
Coordination { channels
conditional variable (CV)
wait group
Deadlock:
{ T1 is waiting T2 to release a lock
T2 is waiting T1 to release a lock
T1: mu1.lock()
T2: mu2.lock() } wait for T2
T1: mu2.lock() wait for T1
T2: mu1.lock()

```
package main
import (
	"fmt"
	"sync"
)
//
// Several solutions to the crawler exercise from the Go tutorial
// https://tour.golang.org/concurrency/10
//
//
// Serial crawler
//
func Serial(url string, fetcher Fetcher, fetched map[string]bool) {
	if fetched[url] {
		return
	}
	fetched[url] = true
	urls, err := fetcher.Fetch(url)
	if err != nil {
		return
	}
	for _, u := range urls {
		Serial(u, fetcher, fetched)
	}
	return
}
//
// Concurrent crawler with shared state and Mutex
//
type fetchState struct {
	mu	sync.Mutex
	fetched map[string]bool
}
func ConcurrentMutex(url string, fetcher Fetcher, f *fetchState) {
	f.mu.Lock()
	already := f.fetched[url]
	f.fetched[url] = true
	f.mu.Unlock()
if already {
		return
	}
urls, err := fetcher.Fetch(url)
	if err != nil {
		return
	}
	var done sync.WaitGroup
	for _, u := range urls {
		done.Add(1)
    u2 := u
		go func() {
			defer done.Done()
			ConcurrentMutex(u2, fetcher, f)
		}()
		//go func(u string) {
		//	defer done.Done()
		//	ConcurrentMutex(u, fetcher, f)
		//}(u)
	}
	done.Wait()
	return
}
func makeState() *fetchState {
	f := &fetchState{}
	f.fetched = make(map[string]bool)
	return f
}
//
// Concurrent crawler with channels
//
func worker(url string, ch chan []string, fetcher Fetcher) {
	urls, err := fetcher.Fetch(url)
	if err != nil {
		ch <- []string{}
	} else {
		ch <- urls
	}
}
func master(ch chan []string, fetcher Fetcher) {
	n := 1
	fetched := make(map[string]bool)
	for urls := range ch {
		for _, u := range urls {
			if fetched[u] == false {
				fetched[u] = true
				n += 1
				go worker(u, ch, fetcher)
			}
		}
		n -= 1
		if n == 0 {
			break
		}
	}
}
func ConcurrentChannel(url string, fetcher Fetcher) {
	ch := make(chan []string)
	go func() {
		ch <- []string{url}
	}()
	master(ch, fetcher)
}
//
// main
//
func main() {
	fmt.Printf("=== Serial===\n")
	Serial("http://golang.org/", fetcher, make(map[string]bool))
fmt.Printf("=== ConcurrentMutex ===\n")
	ConcurrentMutex("http://golang.org/", fetcher, makeState())
fmt.Printf("=== ConcurrentChannel ===\n")
	ConcurrentChannel("http://golang.org/", fetcher)
}
//
// Fetcher
//
type Fetcher interface {
	// Fetch returns a slice of URLs found on the page.
	Fetch(url string) (urls []string, err error)
}
// fakeFetcher is Fetcher that returns canned results.
type fakeFetcher map[string]*fakeResult
type fakeResult struct {
	body string
	urls []string
}
func (f fakeFetcher) Fetch(url string) ([]string, error) {
	if res, ok := f[url]; ok {
		fmt.Printf("found:  %s\n", url)
		return res.urls, nil
	}
	fmt.Printf("missing: %s\n", url)
	return nil, fmt.Errorf("not found: %s", url)
}
// fetcher is a populated fakeFetcher.
var fetcher = fakeFetcher{
	"http://golang.org/": &fakeResult{
		"The Go Programming Language",
		[]string{
			"http://golang.org/pkg/",
			"http://golang.org/cmd/",
		},
	},
	"http://golang.org/pkg/": &fakeResult{
		"Packages",
		[]string{
			"http://golang.org/",
			"http://golang.org/cmd/",
			"http://golang.org/pkg/fmt/",
			"http://golang.org/pkg/os/",
		},
	},
	"http://golang.org/pkg/fmt/": &fakeResult{
		"Package fmt",
		[]string{
			"http://golang.org/",
			"http://golang.org/pkg/",
		},
	},
	"http://golang.org/pkg/os/": &fakeResult{
		"Package os",
		[]string{
			"http://golang.org/",
			"http://golang.org/pkg/",
		},
	},
}
```

Cons: it could create too many threads if too many URLs.

make these 2 lines atomic

This parameter need to be passed because there is a for loop before and we want to get the same u as in the loop.

check "race" by running: "go -race ... go" race detector.