```
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()
	if f.fetched[url] {
		f.mu.Unlock()
		return
	}
	f.fetched[url] = true
	f.mu.Unlock()

	urls, err := fetcher.Fetch(url)
	if err != nil {
		return
	}
	var done sync.WaitGroup
	for _, u := range urls {
		done.Add(1)
		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/",
		},
	},
}
```