University of Stavanger

Faculty of Science and Technology

Exam in subject:	**DAT320 Operating Systems**
Date:	**December 4, 2019**
Duration:	**4 hours**
Allowed remedies:	**None**
The exam consists of:	**4 exercises on 18 pages**
Contact during exam:	**Hein Meling, mobile 924 36 131**
Remarks:	Your responses should be entered into Inspera Assessment. The exam is prepared in both English and Norwegian. English is the primary language, Norwegian is a translation. If there are discrepancies or missing translations, please refer to the English text. For some multiple choice questions there are multiple correct answers; they are explicitly mentioned in the question.
Attachments:	Golang Sync Package API Documentation

Question 1: Operating System Concepts (29/100)

(a) (2%) Hva er et operativsystem? Forklar med en setning.
What is an operating system? Explain with one sentence.

Solution:
An operating system is the software layer that manages a computer's resources for its users and their applications.

(b) (3%) Hva er *virtualisering*? Forklar kort.
What is virtualization? Explain briefly.

Solution:
Virtualization is general technique for sharing a physical resource (processor, memory, disk etc) by transforming it to a more general and easy-to-use virtual form.

(c) (2%) Gi minst to eksempeler på *virtualisering*.
Give at least two examples of virtualization.

Solution:

- Virtualization of the processor with processes.
- Virtualization of memory with address spaces.

(d) (3%) Forklar programvare designprinsippet knyttet til separasjon av mekanisme og policy.
Explain the software design principle of separating mechanism from policy.

Solution:
By separating mechanism and policy, we can provide a generic low-level mechanism (machinery) that can be reused across multiple high-level policy decisions. Such policy decisions can more easily be tuned and adapted to different environments and workloads, without changing the underlying mechanism.

(e) (2%) Gi minst ett eksempel på en mekanisme.
Give at least one example of a mechanism.

Solution:

- Context switching: stop running one process, start running another process.
- Authorization: grant access, deny access.

(f) (2%) Gi minst ett eksempel på en policy.
Give at least one example of a policy.

Solution:

- Scheduling: different scheduling policies can be implemented without changing the underlying context switching mechanism.
- Allocation of resources: based on the authorization mechanism, decide which resources to grant access to.

(g) (3%) Oppgi de tre hovedtilstandene som en prosess kan innta.
Give the three main states that a process can take on.

Solution:
Running, Ready, Blocked = Waiting.

(h) (3%) Oppgi de mulige overgangene mellom de tre tilstandene, og hvordan disse overgangene aktiveres.

Give the possible transitions between the three states, and how these transitions are activated.

Solution:

- Running to Ready: OS preempts the running process.
- Running to Blocked: process makes system call to initiate IO.
- Ready to Running: OS schedules process to run next.
- Blocked to Ready: IO requested by process is done.

(i) (3%) Forklar hvorfor en prosess har tilstand (hva er nytten for operativsystemet.)

Explain why a process has state? (what is the benefit for the operating system.)

Solution:
A process can be in **one** of several states at a time. This is a way for the OS to manage the processes and to facilitate efficient resource utilization. In particular, a Blocked process will not consume CPU resources while waiting for IO to finish, which would otherwise be extremely wasteful.

(j) (3%) Forklar hva en prosess er og hvordan operativsystemet håndterer prosesser med hensyn på minne og kjøring.

Explain what a process is, and how the operating system handles processes with respect to memory and execution.

Solution:
A process is an instance of a program executing with limited privileges.

With respect to memory, a process is divided into two *static components*: code and data, and two *dynamic components*: heap and stack. See Fig. 2.2 in the text book.

In the OS, a process is represented by a Process Control Block (PCB): In-kernel data structure containing per-process state: registers, Stack Pointer (SP), Program Counter (PC), priority, owner, open files, its privileges and so on.

There is two parts to a process:

- Thread: a sequence of instructions within a process
  - Potentially many threads per process
  - Thread = lightweight process (goroutine is even more lightweight)
- Address space
  - Memory that a process can access
  - Other permissions the process has (e.g. which functions it can call, what files it can access)

(k) (3%) Hva er hensikten med minnebeskyttelse? Og forklar en metode som kan benyttes for å oppnå slik minnebeskyttelse.

What is the purpose of memory protection? And explain one method to achieve such memory protection.

Solution:

We need to prevent user code from overwriting the kernel or other processes. So we need limits on memory accesses. The simplest approach is the base and bounds approach:

We can use two HW registers:

- *base*: start of a process's memory region, and
- *bounds*: the process's length
- can support variable sized chunks per process
- can only be changed by privileged instructions (in kernel mode)

Every time the CPU fetches an instruction or data, the PC or address reference is checked against the range $[base, base + bounds]$. If in range: Proceed, otherwise: Exception

Question 2: Concurrency (22/100)

(a) (5%) Det er en *race condition* i koden nedenfor. Forklar hvor og hvorfor det er en race condition i koden. Foreslå en løsning for å unngå den.

There is a race condition in the code below. Explain where and why? Propose a solution to avoid the race condition.

```go
 1 package main
 2
 3 import (
 4   "flag"
 5   "fmt"
 6   "log"
 7   "sync"
 8 )
 9
10 func main() {
11   var (
12     concurrency = flag.Int("concurrency", 10, "number␣of␣goroutines␣to␣run")
13     race        = flag.Bool("race", false, "run␣with␣race␣condition")
14   )
15   flag.Parse()
16   if *concurrency < 2 {
17     log.Fatalf("Cannot␣demonstrate␣race␣condition␣with␣fewer␣than␣2␣goroutines")
18   }
19   if *race {
20     withRace(*concurrency)
21   } else {
22     withoutRace(*concurrency)
23   }
24 }
25
26 func withRace(concurrency int) {
27   var wg sync.WaitGroup
28   wg.Add(concurrency)
29   fmt.Printf("Creating␣%d␣goroutines\n", concurrency)
30   for i := 0; i < concurrency; i++ {
31     go func() {
32       fmt.Printf("%d␣", i)
33       wg.Done()
34     }()
35   }
36   wg.Wait()
37   fmt.Println("\nDone...")
38 }
```

Solution:

There is a race condition because the various anonymous goroutines created in the loop reads the variable i in line 32, while it is being updated by the main goroutine.

To fix it we must pass i as an argument to the goroutine function, as shown in the code below. Note that, to distinguish the loop variable i from the goroutine local variables, we can rename the variable as we pass it into the goroutine function. Specifically, here we use x as the goroutine local variable; each goroutine gets its own copy of x, instead of sharing a variable with the main goroutine.

```go
1 package main
2
```

```go
import (
  "fmt"
  "sync"
)

func withoutRace(concurrency int) {
  var wg sync.WaitGroup
  wg.Add(concurrency)
  fmt.Printf("Creating %d goroutines\n", concurrency)
  for i := 0; i < concurrency; i++ {
    go func(x int) {
      fmt.Printf("%d ", x)
      wg.Done()
    }(i)
  }
  wg.Wait()
  fmt.Println("\nDone...")
}
```

(b) (10%) Bruk koden nedenfor for å implementere funksjonen `ParallelWordCount()` for å beregne totalt antall ord i `input`.

Use the code provided below to implement the function `ParallelWordCount()` to compute the total number of words in the `input`.

```go
package lab4

import "unicode"

func wordCount(b []byte) (words int) {
  inword := false
  for _, v := range b {
    r := rune(v)
    if unicode.IsSpace(r) && inword {
      words++
      inword = false
    }
    inword = unicode.IsLetter(r)
  }
  return
}

func shardSlice(input []byte, numShards int) (shards [][]byte) {
  shards = make([][]byte, numShards)
  if numShards < 2 {
    shards[0] = input[:]
    return
  }
  shardSize := len(input) / numShards
  start, end := 0, shardSize
  for i := 0; i < numShards; i++ {
    for j := end; j < len(input); j++ {
      char := rune(input[j])
      if unicode.IsSpace(char) {
        // split slice at position j, where there is a space
        // note: need to include the space in the shard to get accurate count
        end = j + 1
        shards[i] = input[start:end]
        start = end
        end += shardSize
        break
      }
    }
  }
  shards[numShards-1] = input[start:]
  return
}

func ParallelWordCount(input []byte, numShards int) (words int) {
  // TODO: Implement parallel word count
  return 0
}
```

Solution:

```go
package lab4

```

```go
func chanParallelWordCount(input []byte, numShards int) (words int) {
  cntCh := make(chan int, 1)
  moby := shardSlice([]byte(input), numShards)
  for j := 0; j < numShards; j++ {
    go func(i int) {
      cntCh <- wordCount(moby[i])
    }(j)
  }
  for i := 0; i < numShards; i++ {
    words += <-cntCh
  }
  close(cntCh)
  return
}
```

(c) (2%) Forklar begrepet kritisk region. Når trenger vi å definere en kritisk region?

Explain the term critical section. When do we need to define a critical section?

Solution:

En kritisk region er et kodesegment hvor en prosess gjør endringer på variabler/minne/ressurser som er felles med andre prosesser (eller tråder). Det kan f.eks. være oppdatering av en tabell eller skriving til en fil eller lignende. Vi trenger altså definere en kritisk region når vi har flere prosesser eller tråder som jobber mot den samme ressursen, og trenger å beskytte tilgangen, slik at ulike prosesser/tråder ikke introduserer konflikter.

(d) (2%) Påstand: Operativsystemets lås-implementasjon sikrer deterministisk-ordning i henhold til rekkefølgen som trådene kaller `Lock()`-funksjonen. (Sant/Usant)

Claim: The operating system's lock implementation ensures deterministic ordering according to the order in which the threads called the `Lock()` function. (True/False)

Solution:

False

(e) (3%) Hva er tofaselåsing? Gi eksempel.

What is two-phase locking? Give example.

Solution:

A common way to enforce isolation among transactions is two-phase locking, locking which divides a transaction into two phases. During the expanding phase, locks may be acquired but not released. Then, in the contracting phase, locks may be released but not acquired. In the case of transactions, because we want isolation and durability, the second phase must wait until after the transaction commits or rolls back so that no other transaction sees updates that later disappear. Two phase locking ensures a strong form of isolation called serializability.

Serializablity across transactions ensures that the result of any execution of the program is equivalent to an execution in which transactions are processed one at a time in some sequential order. So, even if multiple transactions are executed concurrently, they can only produce results that they could have produced had they been executed one at a time in some order. Although acquiring multiple locks in arbitrary orders normally risks dead- lock, transactions provide a simple solution. If a set of transactions deadlocks, one or more of the transactions can be forced to roll back, release their locks, and restart at some later time.

Question 3: Scheduling (22/100)

Betrakt følgende sett med prosesser, med ankomst tid og lengden på CPU burst:

Consider the following set of processes, with arrival time and burst time:

Process	Arrival time	Burst Time
P_1	0	5
P_2	1	7
P_3	3	4

(a) (2%) Anta en ikke-avbrytbar korteste job først (SJF) fordelingsalgoritme benyttes. Hva blir ferdigstillingsrekkefølgen for prosessene?

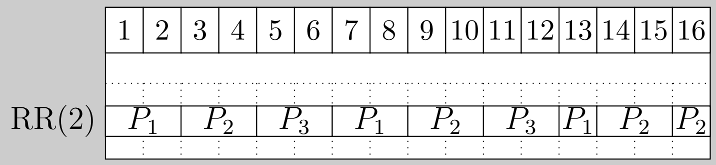
Assuming non-preemptive Shortest Job First scheduling algorithm is used. What is the completion order of the processes?

Solution:
P_1, P_3, P_2

(b) (5%) Tegn et Gantt diagram for kjøring av disse prosessene med Round-Robin fordelingsalgoritmen med tidskvantum $q = 2$, heretter kalt RR(2).

Draw a Gantt chart for the execution of these processes for a Round-Robin scheduling algorithm with time quantum $q = 2$; hereafter called RR(2).

Solution:
Gantt diagram:



(c) (4%) Anta at operativsystemet implementerer en RR(2) fordelingsalgoritme. Hvor mange kontekstbytter er nødvendig for den spesifiserte arbeidslasten? Du må telle den først prosessens kontekstbytte, men ikke den siste, når alle er ferdige.

Assume the operating system implements a RR(2) scheduling algorithm. How many context switches are needed for the specified workload? You should count the first process context switch, but not the last one, when all processes are done.

Solution:
9. We include the first context switch to bring in P_1, but not the last one when P_2 is done.

(d) (2%) Anta RR(2) fordelingsalgoritmen. Hva blir den gjennomsnittlige omløpstiden for de tre prosessene?

Assume RR(2) scheduling. What is the average turnaround time for the three processes?

Solution:
Turnaround time = Completion time - Arrival time

$$
\begin{aligned}
T_t &= T_c - T_a \\
T_t(P_1) &= 13 - 0 = 13 \\
T_t(P_2) &= 16 - 1 = 15 \\
T_t(P_3) &= 12 - 3 = 9 \\
\overline{T_t} &= (13 + 15 + 9)/3 = 12.33
\end{aligned}
$$

(e) (2%) Anta RR(2) fordelingsalgoritmen. Hva blir den gjennomsnittlige responstiden for de tre prosessene?

Assume RR(2) scheduling. What is the average response time for the three processes?

Solution:
Response time = First run - Arrival time

$$
\begin{aligned}
T_r &= T_f - T_a \\
T_r(P_1) &= 0 - 0 = 0 \\
T_r(P_2) &= 2 - 1 = 1 \\
T_r(P_3) &= 4 - 3 = 1 \\
\overline{T_t} &= (0 + 1 + 1)/3 = 2/3 = 0.667
\end{aligned}
$$

(f) (3%) Anta RR(2) fordelingsalgoritmen. Hva blir den gjennomsnittlige ventetiden for de tre prosessene? Ventetiden er omløpstiden minus bursttiden.

Assume RR(2) scheduling. What is the average waiting time for the three processes? Waiting time is the turnaround time minus the burst time.

Solution:
Waiting time = Completion time - Arrival time - Burst time

$$
\begin{aligned}
T_w &= T_c - T_a - T_b & &\text{Waiting time observed from Gantt} \\
T_w(P_1) &= 13 - 0 - 5 = 8 & &4 + 4 \\
T_w(P_2) &= 16 - 1 - 7 = 8 & &1 + 4 + 3 \\
T_w(P_3) &= 12 - 3 - 4 = 5 & &1 + 4 \\
\overline{T_w} &= (8 + 8 + 5)/3 = 7
\end{aligned}
$$

(g) (2%) Hvilken av disse fordelingsalgoritmene er ikke-avbrytbar?

Which one of the following scheduling algorithms is non-preemptive?

A. Multilevel Queue Scheduling
B. Multilevel Feedback Queue Scheduling
C. Round Robin
D. First Come First Served
E. None of the above

(h) (2%) Den mest optimale fordelingsalgoritmen er:

The most optimal scheduling algorithm is:

A. First Come First Serve

B. Round Robin

C. Shortest Job First

D. None of the above

Question 4: Caching (27/100)

(a) (5%) Forklar begrepene tidsmessig lokalitet og romlig lokalitet og hvordan disse er relatert til caching.

Explain the terms temporal locality and spatial locality and how these are related to caching.

Solution:

The terms are related to caching in that they represent sources of predictability that we can leverage to get good performance for caching.

Temporal locality: programs tend to reference the same instructions and data that they have recently accessed (time dimension). For example, instructions inside a loop or a data structure that is repeatedly accessed. By caching these memory values, we can improve performance.

Spatial locality: programs tend to reference data near other data that has been recently referenced (space dimension). For example, the next instruction to execute is usually near to the previous one, and different fields in the same data structure tend to be referenced at nearly the same time. To exploit this, caches are often designed to load a block of data at the same time, instead of a single location.

(b) (5%) Forklar kort hva arbeidssettmodellen forsøker å modellere.

Explain briefly what the working set model is attempting to model.

Solution:

The working set model attempts to model the critical mass of a program's memory pages, such that most memory references will result in a cache hit, and thus improve the program's performance.

(c) (5%) Anta at en nylig opprettet prosess har fått tildelt 4 siderammer, og at den deretter genererer sidereferansene som indikert nedenfor. Hvor mange cache misses observererers ved bruk av FIFO side-erstattningsalgoritmen?

Consider a newly-created process that has been allocated 4 page frames, and then generates the page references as indicated below. How many cache misses are observed with the use of the FIFO page replacement algorithm?

Page references:

`CEAFCCGCFEDBCDEAGADF`

Solution:

12.

```
C E A F C C G C F E D B C D E A G A D F
C       + + G         B
  E           C         +     A   +
    A             E         +   G
      F         +   D     +         + F

Cache Size = 4, Cache hits = 8, misses = 12
```

(d) (5%) Denne side referansestrømmen viser seg å være svært vanlig i systemet, og for å forbedre ytelsen ønsker OS leverandøren å øke cache størrelsen til 5 siderammer. Hvor mange cache misses observeres ved bruk av den nye cache størrelsen?

This page reference stream turns out to be very common in the system, and to improve performance, the OS vendor increases the cache size to 5 page frames. How many cache misses are observed with the new cache size?

Solution:

13.

```
C E A F C C G C F E D B C D E A G A D F
C       + +   +     D     +     G
  E               +   B             D
    A                   C             F
      F         +           E
            G                 A   +

Cache Size = 5, Cache hits = 7, misses = 13
```

(e) (2%) Forklar kort cache oppførselen observert i de to foregående oppgavene.

Explain briefly the cache behavior observed in the two preceding exercises.

Solution:

We observed Belady's anomaly.

(f) (5%) OS leverandøren prøver istedet å erstatte FIFO med en minst-ofte (sjeldnest) brukt (LFU) cache, og en cache størrelse på 5 siderammer. Hvor mange cache misses observeres ved bruk av LFU side-erstattningsalgoritmen?

The OS vendor next tries to replace FIFO with a Least Frequently Used (LFU) cache, and cache size of 5 page frames. How many cache misses are observed with the LFU page replacement algorithm?

Solution:

10. Note that when multiple entries in the LFU cache have equal frequency (use count), the traces may be different, and likewise the miss count. To get deterministic result, we always use the lowest or highest cache position when replacing pages with equal count. For this particular trace, both approaches result in 10 cache misses. (It is unknown if an otherwise correct trace, using a non-deterministic replacement strategy will result in different cache misses.)

```
C E A F C C G C F E D B C D E A G A D F
C       + +   +         +
  E               +         +
    A               D B   D   A   + D
      F         +                     +
            G                   +

Cache Size = 5, Cache hits = 10, misses = 10

C E A F C C G C F E D B C D E A G A D F
C       + +   +         +
  E               +         +
    A                         +   +
      F         +                     +
            G       D B   D     G   D

Cache Size = 5, Cache hits = 10, misses = 10
```

Package sync

```
import "sync"
```

Overview ▾

Package sync provides basic synchronization primitives such as mutual exclusion locks. Other than the Once and WaitGroup types, most are intended for use by low-level library routines. Higher-level synchronization is better done via channels and communication.

Values containing the types defined in this package should not be copied.

Index ▾

Examples (Expand All)

Package files

cond.go map.go mutex.go once.go pool.go poolqueue.go runtime.go rwmutex.go waitgroup.go

type Cond

Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.

Each Cond has an associated Locker L (often a *Mutex or *RWMutex), which must be held when changing the condition and when calling the Wait method.

A Cond must not be copied after first use.

```
type Cond struct {

    // L is held while observing or changing the condition
    L Locker
    // contains filtered or unexported fields
}
```

func NewCond

```
func NewCond(l Locker) *Cond
```

NewCond returns a new Cond with Locker l.

func (*Cond) Broadcast

```
func (c *Cond) Broadcast()
```

Broadcast wakes all goroutines waiting on c.

It is allowed but not required for the caller to hold c.L during the call.

func (*Cond) Signal

```
func (c *Cond) Signal()
```

Signal wakes one goroutine waiting on c, if there is any.

It is allowed but not required for the caller to hold c.L during the call.

func (*Cond) Wait

```
func (c *Cond) Wait()
```

Wait atomically unlocks c.L and suspends execution of the calling goroutine. After later resuming execution, Wait locks c.L before returning. Unlike in other systems, Wait cannot return unless awoken by Broadcast or Signal.

Because c.L is not locked when Wait first resumes, the caller typically cannot assume that the condition is true when Wait returns. Instead, the caller should Wait in a loop:

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

type Locker

A Locker represents an object that can be locked and unlocked.

```
type Locker interface {
    Lock()
    Unlock()
}
```

type Map 1.9

Map is like a Go map[interface{}]interface{} but is safe for concurrent use by multiple goroutines without additional locking or coordination. Loads, stores, and deletes run in amortized constant time.

The Map type is specialized. Most code should use a plain Go map instead, with separate locking or coordination, for better type safety and to make it easier to maintain other invariants along with the map content.

The Map type is optimized for two common use cases: (1) when the entry for a given key is only ever written once but read many times, as in caches that only grow, or (2) when multiple goroutines read, write, and overwrite entries for disjoint sets of keys. In these two cases, use of a Map may significantly reduce lock contention compared to a Go map paired with a separate Mutex or RWMutex.

The zero Map is empty and ready for use. A Map must not be copied after first use.

```
type Map struct {
    // contains filtered or unexported fields
}
```

func (*Map) Delete 1.9

```
func (m *Map) Delete(key interface{})
```

Delete deletes the value for a key.

func (*Map) Load 1.9

```
func (m *Map) Load(key interface{}) (value interface{}, ok bool)
```

Load returns the value stored in the map for a key, or nil if no value is present. The ok result indicates whether value was found in the map.

func (*Map) LoadOrStore 1.9

```
func (m *Map) LoadOrStore(key, value interface{}) (actual interface{},
loaded bool)
```

LoadOrStore returns the existing value for the key if present. Otherwise, it stores and returns the given value. The loaded result is true if the value was loaded, false if stored.

func (*Map) Range 1.9

```
func (m *Map) Range(f func(key, value interface{}) bool)
```

Range calls f sequentially for each key and value present in the map. If f returns false, range stops the iteration.

Range does not necessarily correspond to any consistent snapshot of the Map's contents: no key will be visited more than once, but if the value for any key is stored or deleted concurrently, Range may reflect any mapping for that key from any point during the Range call.

Range may be O(N) with the number of elements in the map even if f returns false after a constant number of calls.

func (*Map) Store 1.9

```
func (m *Map) Store(key, value interface{})
```

Store sets the value for a key.

type Mutex

A Mutex is a mutual exclusion lock. The zero value for a Mutex is an unlocked mutex.

A Mutex must not be copied after first use.

```
type Mutex struct {
    // contains filtered or unexported fields
}
```

func (*Mutex) Lock

```
func (m *Mutex) Lock()
```

Lock locks m. If the lock is already in use, the calling goroutine blocks until the mutex is available.

func (*Mutex) Unlock

```
func (m *Mutex) Unlock()
```

Unlock unlocks m. It is a run-time error if m is not locked on entry to Unlock.

A locked Mutex is not associated with a particular goroutine. It is allowed for one goroutine to lock a Mutex and then arrange for another goroutine to unlock it.

type Once

Once is an object that will perform exactly one action.

```
type Once struct {
    // contains filtered or unexported fields
}
```

▹ Example

func (*Once) Do

```
func (o *Once) Do(f func())
```

Do calls the function f if and only if Do is being called for the first time for this instance of Once. In other words, given

```
var once Once
```

if once.Do(f) is called multiple times, only the first call will invoke f, even if f has a different value in each invocation. A new instance of Once is required for each function to execute.

Do is intended for initialization that must be run exactly once. Since f is niladic, it may be necessary to use a function literal to capture the arguments to a function to be invoked by Do:

```
config.once.Do(func() { config.init(filename) })
```

Because no call to Do returns until the one call to f returns, if f causes Do to be called, it will deadlock.

If f panics, Do considers it to have returned; future calls of Do return without calling f.

type Pool 1.3

A Pool is a set of temporary objects that may be individually saved and retrieved.

Any item stored in the Pool may be removed automatically at any time without notification. If the Pool holds the only reference when this happens, the item might be deallocated.

A Pool is safe for use by multiple goroutines simultaneously.

Pool's purpose is to cache allocated but unused items for later reuse, relieving pressure on the garbage collector. That is, it makes it easy to build efficient, thread-safe free lists. However, it is not suitable for all free lists.

An appropriate use of a Pool is to manage a group of temporary items silently shared among and potentially reused by concurrent independent clients of a package. Pool provides a way to amortize allocation overhead across many clients.

An example of good use of a Pool is in the fmt package, which maintains a dynamically-sized store of temporary output buffers. The store scales under load (when many goroutines are actively printing) and shrinks when quiescent.

On the other hand, a free list maintained as part of a short-lived object is not a suitable use for a Pool, since the overhead does not amortize well in that scenario. It is more efficient to have such objects implement their own free list.

A Pool must not be copied after first use.

```
type Pool struct {

    // New optionally specifies a function to generate
    // a value when Get would otherwise return nil.
    // It may not be changed concurrently with calls to Get.
    New func() interface{}
    // contains filtered or unexported fields
}
```

▷ Example

func (*Pool) Get 1.3

```
func (p *Pool) Get() interface{}
```

Get selects an arbitrary item from the Pool, removes it from the Pool, and returns it to the caller. Get may choose to ignore the pool and treat it as empty. Callers should not assume any relation between values passed to Put and the values returned by Get.

If Get would otherwise return nil and p.New is non-nil, Get returns the result of calling p.New.

func (*Pool) Put 1.3

```
func (p *Pool) Put(x interface{})
```

Put adds x to the pool.

type RWMutex

A RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. The zero value for a RWMutex is an unlocked mutex.

A RWMutex must not be copied after first use.

If a goroutine holds a RWMutex for reading and another goroutine might call Lock, no goroutine should expect to be able to acquire a read lock until the initial read lock is released. In particular, this prohibits recursive read locking. This is to ensure that the lock eventually becomes available; a blocked Lock call excludes new readers from acquiring the lock.