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 12 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.

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

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

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

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

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

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

(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.

(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.)

(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.

(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.

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.

```
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 }
```

(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`.

```
1  package lab4
2
3  import "unicode"
4
5  func wordCount(b []byte) (words int) {
6    inword := false
7    for _, v := range b {
8      r := rune(v)
9      if unicode.IsSpace(r) && inword {
10       words++
11       inword = false
12     }
13     inword = unicode.IsLetter(r)
14   }
15   return
16 }
17
18 func shardSlice(input []byte, numShards int) (shards [][]byte) {
19   shards = make([][]byte, numShards)
20   if numShards < 2 {
21     shards[0] = input[:]
22     return
23   }
24   shardSize := len(input) / numShards
25   start, end := 0, shardSize
26   for i := 0; i < numShards; i++ {
27     for j := end; j < len(input); j++ {
28       char := rune(input[j])
29       if unicode.IsSpace(char) {
30         // split slice at position j, where there is a space
31         // note: need to include the space in the shard to get accurate count
32         end = j + 1
33         shards[i] = input[start:end]
34         start = end
35         end += shardSize
36         break
37       }
38     }
39   }
40   shards[numShards-1] = input[start:]
41   return
42 }
43
44 func ParallelWordCount(input []byte, numShards int) (words int) {
45   // TODO: Implement parallel word count
46   return 0
47 }
```

(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?

(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)*

(e) (3%) Hva er tofaselåsing? Gi eksempel.
What is two-phase locking? Give example.

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?

(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).

(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.

(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?

(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?

(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.

(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.

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

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

(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 observerers 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`

(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?

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

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

(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?

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.