University of Stavanger

Faculty of Science and Technology

Exam in subject:	**DAT320 Operating Systems**
Date:	**December 4, 2018**
Duration:	**4 hours**
Allowed remedies:	**None**
The exam consists of:	**8 exercises on 23 pages**
Contact during exam:	**Vinay Setty, tlf. 518 32760**
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 questions there is a negative point of -1 if your answer is wrong. Unless explicitly mentioned about the negative score, the minimum score you can get for a question is 0. For some multiple choice questions there are multiple correct answers they are explicitly mentioned in the question.
Attachments:	Golang Sync Package API Documentation and lab 6 (Zap lab) description

---

Question 1: Operating System Concepts (15/100)

(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%) Hvilke roller har et operativsystem? Gi eksampler.

What roles does an operating system play? Give examples.

Solution:

We can think of an operating system as playing three different roles:

1. Referee: Operating systems manage resources shared between different applications running on the same physical machine. Examples: Memory protection, scheduler
2. Illusionist: Operating systems provide an abstraction of physical hardware to simplify application design. Examples: Virtual memory, file system
3. Glue: Operating systems provides a set of common services that facilitate sharing among applications. Examples: HAL, system calls, copy paste

(c) (5%) Hva er hensikten med dual-mode operasjon på en CPU?

What is the purpose of dual-mode operation on a CPU?

Solution:

There are two modes of operation on a CPU: Kernel mode and user mode. The purpose of the two modes is to limit the privileges of a user program, so that it cannot cause harm to the machine, operating system, or other user processes running on the system.

- Kernel mode
  - Execution with the full privileges of the hardware
  - Can change privilege level
  - Can disable/enable interrupts
  - R/W from/to any memory, any I/O device, any disk sector
- User mode
  - Limited privileges
  - Only those (privileges) granted by the kernel

(d) (5%) Hvor mange nye prosesser blir opprettet når følgende C-program utføres? (inklusiv hovedprosessen) Advarsel: Dette spørsmålet har negativ poengsum på -1 for feil svar. *How many new processes are created when the following C program is executed? (including the main process). Warning: This question has negative score of -1 for an incorrect answer.*

Advarsel: Dette spørsmålet har negativ poengsum på -1 for feil svar. Warning: This question has negative score of -1 for an incorrect answer.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  void forkthem(int n){
4    if(n % 2 == 0){
5      int pid = fork();
6    }
7    if(n > 0)
8      forkthem(n-1);
9  }
10 int main(int argc, char** argv){
11   forkthem(5);
12 }
```

Solution:

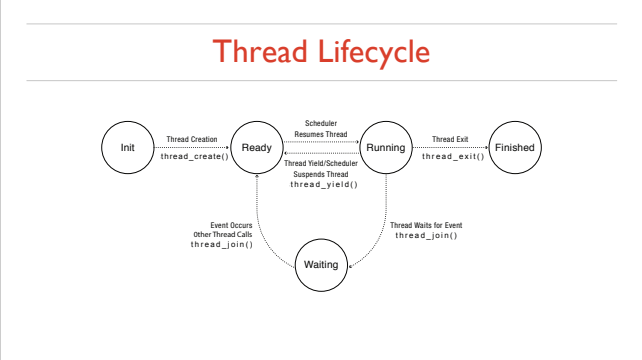
Fork is executed only when n is even so that is 3 times and $2^3 - 1 = 7$(forks) + 1 (main) = 8

Question 2: Multithreading (10/100)

(a) (4%) Forklar de forskjellige tilstandene i en tråds livssyklus og de ulike systemkall som trengs for å bytte fra en tilstand til en annen?

Explain the different states in the lifecycle of a thread and the different system calls needed to transition from one state to another?

Solution:



(b) (3%) Forklar stegene involvert i mekanismen for tråd kontektsbytte.

Explain the steps involved in the mechanism for thread context switching.

Solution:

A thread context switch involves suspending the execution of the currently running thread and resume execution of some other thread. This is done by copying the currently running thread's registers from the processor to the thread's TCB and then copying the other thread's registers from that thread's TCB to the processor.

Suspending a thread involves a processor mode switch (user mode to kernel mode), which can be triggered by an interrupt, an exception, or a system call trap. A kernel handler must be set up to handle the context switching described above (copying to/from TCB and processor registers).

(c) (3%) Forklar asynchronous I/O med et eksempel og hva er som fordeler og ulemper med async I/O over tråder.

Explain asynchronous I/O with an example and it's advantages and disadvantages over multithreading.

Solution:

I/O processing that permits other processing to continue before the transmission has finished.

- Single-threaded program issue multiple concurrent I/O requests.
- Process makes system call to issue I/O request
- Returns immediately
- Ways to get result from I/O request
    1. OS can call signal handler (interrupt)
    2. Place result in process's memory
    3. Store result in kernel until process makes other system call to get result

example: aio_read

Advantages

- Performance: Coping with high-latency I/O in single core processor systems

- Both can overlap I/O with computation
- Space and context switch overhead is lower
- No user locks necessary OS takes care of it

Disadvantages

- If the thread system provided by the OS is efficient there is not much benefit from Async IO
- asynchronous I/O cannot leverage multiprocessor systems

Question 3: Concurrency (15/100)

(a) (10%) Kanaler i Go er "rør" som kan brukes til å kommunisere data mellom goroutiner. Du kan sende verdier over en kanal i en goroutine og motta disse verdiene i en annen goroutine.

Skriv koden i Go for å implementere oppførselen til en *buffered kanal* ved hjelp av en kødatastruktur og betingelsesvariabler. Implementer de to metodene "WriteToChannel" og "ReadFromChannel" i kodemalen nedenfor, for å etterligne '←' operatoren for kanaler i Go. Du kan referere til dokumentasjonen for Go Sync package i vedlegget.

Channels in Go are pipes used for communicating data between goroutines. You can send values on a channel in one goroutine and receive those values in another goroutine.

Write the Go code to implement the behavior of a buffered channel *using a queue data structure and condition variables. Implement the two methods "WriteToChannel" and "ReadFromChannel" in the code snippet below to mimic the '←' operator for channels in Go. You may refer to the Go Sync package documentation in the appendix.*

```
1  import "sync"
2
3  type Channel struct {
4    // State variables
5    buffer_size int
6    buffer     [buffer_size]int
7    front     int
8    nextEmpty int
9  }
10
11 func (c *Channel) WriteToChannel(item int) {
12
13 }
14
15 func (c *Channel) ReadFromChannel() (item int) {
16
17 }
```

Solution:

```
1  import "sync"
2
3  // Channel is implemented as a Bounded Blocking Queue using two condition variables
4  type Channel struct {
5    // Synchronization variables
6    lock        sync.Mutex
7    itemAdded   sync.Cond
8    itemRemoved sync.Cond
9    // State variables
10   buffer     [buffer_size]int
11   front     int
12   nextEmpty int
13 }
14
15 func (q *Channel) WriteToChannel(item int) {
```

```
16    q.lock.Lock()
17    for q.nextEmpty-q.front == buffer_size {
18      q.itemRemoved.Wait() // queue is full, wait for remove
19    }
20    q.buffer[q.nextEmpty%buffer_size] = item
21    q.nextEmpty++
22    q.itemAdded.Signal()
23    q.lock.Unlock()
24  }
25
26  func (q *Channel) ReadFromChannel() (item int) {
27    q.lock.Lock()
28    for q.front == q.nextEmpty {
29      q.itemAdded.Wait() // queue is empty, wait for add
30    }
31    item = q.buffer[q.front%buffer_size]
32    q.front++
33    q.itemRemoved.Signal()
34    q.lock.Unlock()
35  }
```

(b) (5%) Skriv Go kode for å implementere en trådsikker Bounded Blocking Queue (en kø med begrenset størrelse) med bruk av Go kanaler. Er dette ekvivalent med å bruke betingelsesvariabler? Forklar.

Write Go code for implementing a thread-safe Bounded Blocking Queue (a queue with limited size) using Go channels. Is it equivalent to using condition variables? Explain.

Solution:

```
1   package tsqueue
2
3   // ChQueue is a Bounded Blocking Queue
4   type ChQueue struct {
5     // State variables
6     items chan int
7   }
8
9   func NewChQueue() *ChQueue {
10    return &ChQueue{items: make(chan int, max)}
11  }
12
13  func (q *ChQueue) insert(item int) {
14    q.items <- item
15  }
16
17  func (q *ChQueue) remove() (item int) {
18    return <-q.items
19  }
```

Yes it is equivalent behavior in terms of waiting if the buffer is full, but in some aspects the behavior could differ like when a signal is called on condition variable, the order of next thread which gets to hold enter the critical section may differ.

Question 4: Advanced Synchronization (10/100)

(a) (5%) Hva er en vranglås? Hva er de nødvendige betingelsene for at det skal oppstå vranglås?

What is a deadlock? What are the necessary conditions for deadlock*?*

Solution:

A deadlock is a condition in which none of the threads/processes can proceed because they are waiting for resoures held by other threads/processes each other resulting in a cyclic dependency.

1. Limited access to resources (infinite amount resources: no deadlock)
2. No preemption (If some has a resource, system can't take it back)
3. Waiting while holding (multiple independent requests). A thread holds a resource while waiting for another.
4. Circular waiting. There is a set of waiting threads such that each thread is waiting for a resource held by another

(b) (5%) Anta at vi har n tråder, $t_1, \ldots, t_n$ som deler m identiske ressurser, som kan reserveres og frigjøres en om gangen. Maksimalt antall ressurser som en tråd t_i til enhver tid kan reservere, er gitt ved S_i, hvor $S_i > 0$. Hvilke av følgende betingelser må være sanne for å unngå at vranglås oppstår?

Assume there are n threads, $t_1, \ldots, t_n$ sharing m identical resources, which can be acquired and released one at a time. The maximum number of resources a thread t_i acquires at any given time is represented as S_i, where $S_i > 0$. Which of the following conditions must be true to avoid that deadlock occurs?

1. $\forall i, S_i < m$
2. $\forall i, S_i < n$
3. $\sum_{i=1}^{n} S_i < m + n$
4. $\sum_{i=1}^{n} S_i < m \cdot n$

Solution:

In the extreme condition, all processes acquire $S_i - 1$ resources and need 1 more resource to avoid deadlock. So the following condition must be true to make sure that deadlock never occurs.

$\sum_{i=1}^{n} S_i - 1 < m$ which can be simplified as $\sum_{i=1}^{n} S_i < m + n$

Question 5: Scheduling (10/100)

(a) (4%) Hvilke av følgende utsagn er sanne? Forutsatt at alle tidsplanleggingsalgoritmene er *preemptive*.

- First-Come, First-Serve (FCFS)
- Shortest-Remaining-Job-First (SRJF)
- Round-Robin (RR)

1. SRJF tidsplanlegging kan føre til utsulting
2. Max-Min rettferdighet kan føre til utsulting
3. RR er alltid bedre enn FCFS når det gjelder gjennomsnittlig svartid
4. FCFS har alltid minst antall kontekstbytter

Which of the following statements are true? Assuming all scheduling algorithms are preemptive.

1. *SRJF scheduling may cause starvation*
2. *Max-Min fairness may cause starvation*
3. *RR is always better than FCFS in terms of average response time*
4. *FCFS always has the least number of context switches*

Solution:
1 and 4 are true. 3 is false because Round robin with a very high quantum is same as FCFS. 2 is false because Max-Min avoids starvation.

(b) (6%) Betrakt følgende sett med prosesser, angitt med lengden på CPU burst og ankomst tid:

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

Process	Burst time	Arrival Time
P_1	5	0
P_2	7	1
P_3	2	2
P_4	1	5

Betrakt følgende sett med *preemptive* tidsplanleggingsalgoritmer:

Consider the following set of preemptive scheduling algorithms:

- First-Come, First-Serve (FCFS)
- Shortest-Remaining-Job-First (SRJF)
- Round-Robin with time quantum=3 (RR)

Responsetiden til en prosess er den tiden det tar for en prosess for å fullføre jobben fra ankomsttidspunktet. *The response time of a process is the time it takes for a process to finish the job from the time of arrival.*

Fill in the values below:

1. ___ har minst gjennomsnittlig svartid
2. SRJF har ___ (antall) kontekstbytter (se bort fra kontekstbytte ved tid 0 for å starte P_1)
3. Svarstid for P_4 ved bruk av RR med tidskvantum 3 er ____

4. Gjennomsnittlig svartid for FCFS er ____
5. Svarstid på P_2 ved bruk av SRJF er ____
6. Første prosessen til å fullføre ved bruk av SRJF er ____

Fill in the values below:

1. *___ has the least average response time*
2. *SRJF has ___ number of context switches (ignore the context switch at time 0 to start P_1)*
3. *Response time of P_4 using RR with quantum 3 is ____*
4. *Average response time of FCFS is ____*
5. *Response time of P_2 using SRJF is ____*
6. *First process to finish using SRJF is ____*

Solution:

Table 1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
FCFS	P1	P1	P1	P1	P1	P2	P2	P2	P2	P2	P2	P2	P3	P3	P4
SRJF	P1	P1	P3	P3	P1	P4	P1	P1	P2	P2	P2	P2	P2	P2	P2
RR	P1	P1	P1	P2	P2	P2	P3	P3	P4	P1	P1	P2	P2	P2	P2
	P1	P2	P3	P4	RT										
FCFS	5	11	12	10	9,5										
SRJF	8	14	2	1	6,25										
RR	11	14	6	4	8,75										

1. SRJF
2. 5
3. 4
4. 9.5
5. 14
6. P_3

Question 6: Caching (15/100)

(a) (3%) Hva er en "cache"?
What is a cache?
Select one or more alternatives:

1. Full copy of physical memory
2. CPU registers
3. Copy of result of computation that can be accessed quickly
4. Recomputing result of operations every time
5. Partial copy of data that can be accessed quickly

Solution:
Two answers are correct:

- Copy of result of computation that can be accessed quickly
- Partial copy of data that can be accessed quickly

(b) (3%) Hvilke utsagn beskriver Zipfs lov?
Which statement describe Zipf's law?
Select one or more alternatives:

1. Frequency of k-th most popular item is proportional to k^{α}
2. Frequency of k-th most popular item is inversely proportional to k^{α}
3. Zipf curve has a heavy tail
4. Zipf curve is linear
5. Zipf curve is growing with increasing number of items

Solution:
Two answers are correct:

- Frequency of k-th most popular item is inversely proportional to k^{α}
- Zipf curve has a heavy tail

(c) (3%) Belady's Anomaly
Når inntreffer Beladys "avvik"?
Belady's anomaly occurs when.
Select an alternative:

1. The number of cache hits grows with the number of cache slots
2. The number of cache misses increase despite adding more cache slots
3. The demand for memory pages is higher than can fit in cache
4. A page is loaded into cache, but is not reused before it is replaced again

Solution:
The number of cache misses increase despite adding more cache slots

(d) (6%) Anta at en nylig opprettet prosess har fått 3 siderammer allokert til seg, og at den deretter genererer sidereferansene indikert nedenfor.
Suppose that a newly-created process has 3 page frames allocated to it, and then generates the page references indicated below.

Page references:

`ABCBADACEBEFBEFBA`

1. How many page faults would occur with FIFO page replacement?
2. How many page faults would occur with LRU page replacement?
3. How many page faults would occur with LFU page replacement?
4. How many page faults would occur with OPT page replacement?

FIFO

	A	B	C	B	A	D	A	C	E	B	E	F	B	E	F	B	A
1	A				+	D				B			+			+	
2		B		+			A					F			+		
3			C					+	E		+			+			A

LRU

	A	B	C	B	A	D	A	C	E	B	E	F	B	E	F	B	A
1	A				+		+			B			+			+	
2		B		+				C				F			+		
3			C			D			E		+			+			A

LFU

	A	B	C	B	A	D	A	C	E	B	E	F	B	E	F	B	A
1	A				+		+										+
2		B		+				C		B		F	B		F	B	
3			C			D			E		+			+			

OPT

	A	B	C	B	A	D	A	C	E	B	E	F	B	E	F	B	A
1	A				+		+					F			+		A
2		B		+		D			E		+			+			
3			C					+		B			+			+	

	Page faults	Cache hits	Check
FIFO	9	8	17
LRU	9	8	17
LFU	11	6	17
OPT	8	9	17

Solution:

1. FIFO 9
2. LRU 9
3. LFU 11
4. OPT 8

Question 7: Security (15/100)

(a) (3%) Hvor i minne inntreffer vanligvis buffer overflyt?
Where in memory does buffer overflows usually happen?
Select an alternative:

1. On the heap
2. On the stack
3. In the kernel
4. In the code/text segment
5. In the data segment

Solution:
Stack. There was a mistake in the digital exam for this course. The scores are manually added to fix the error. There were 2 choices with "on the stack" in digital exam. If student has selected Stack full 3 points should be give. But if there are other options selected in addition, it attracts negative points.

(b) (3%) For å kunne utnytte en buffer overflyt sårbarhet for å oppnå root (superbruker) tilgang til en maskin, må det sårbare programmet være:

To exploit a buffer overflow vulnerability to gain root (super user) access on a machine, the vulnerable program must be:

Select an alternative:

1. a SUID program
2. a GUI program
3. a command line program
4. a GUID program
5. a passwd program

Solution:
A SUID program runs with super user (root) privileges even though it is being run by an non-root user.

(c) (3%) Hvordan kan angriperen nå den ondsinnede koden sin, når adressene kan endres litt mellom hver kjøring av det sårbare programmet?

How can an attacker's exploit code be reached, when the addresses may move around slightly for each execution of the vulnerable program?

Select an alternative:

1. Do a binary search for the exact address of the exploit code
2. Prefix the exploit code with jmp instructions
3. Prefix the exploit code with noop instructions
4. Pad the exploit code with 0x00 values
5. Pad the exploit code with 0xFF values

Solution:
Prefix the exploit code with noop instructions

(d) (3%) Hvis du lager et sterkt passord, med en tilfeldig kombinasjon av tegn, som bare kan angripes ved hjelp av et uttømmende søk, hvor langt må passordet minimum være?

(Spørsmålet skal besvares i henhold til informasjon per juli 2016, når video "How to Choose a Password" ble laget.)

If you choose a strong password, with a random combination of characters, that can only be brute-forced, how long must it be to be secure?

(This question should be answered as per July 2016, when the How to Choose a Password video was made.)

Fill in: ___

Solution:
Nine (9)

(e) (3%) Når du skal lage et passord basert på en kombinasjon av "enkle å huske" ord, hvilke av disse strategiene vil styrke passordet?

When choosing a password based on a combination of "easy to remember" words, which of these strategies help to strengthen the password?

Select an alternative:

1. Pick frequently used words
2. Pick only long frequently used words
3. Pick at least one uncommon word
4. Pick randomly generated (unreadable) words
5. Replace some letters like 'O' with '0' and 'E' with '3'

Solution:
Pick at least one uncommon word (or use a made up word that is not in the dictionary). Frequently used words, whether long or short, will be the first ones to be tried in a dictionary attack. Randomly generated words are not "easy to remember". Similarly, replacing letters with obviously similar numbers is a common trick known to password crackers, and it would be relatively easy to do such replacements on dictionary words.

Question 8: Advanced Synchronization: ZapLab (10/100)

For dette spørsmålet, se ZapLab øvelsen i Vedlegg A hvis det er nødvendig.

For this question, please refer to the ZapLab exercise in Appendix A if necessary.

(a) (10%) Et annonseselskap ønsker å finne topp 3 kanaler som sees av hver bruker som er unikt identifisert av deres ip-adresse for å målrette annonser for dem (i stedet for de mest viste kanalene som var en del av lab 6). Foreslå datastrukturer og en låsestrategi som muliggjør effektiv, parallell og trådsikker beregning av de øverste kanalene som er sett av en gitt bruker. Anta at det er flere goroutiner som behandler zap-hendelsene og oppdaterer datastrukturen og flere goroutiner som beregner topp 3 kanaler for hver bruker. Gi Go-kode eller pseudokode og forklar dine designbeslutninger og avvik som du vurderer.

An advertisement company wants to find top-3 channels watched by each user uniquely identified by their ip address to target ads for them (instead of top viewed channels which was part of lab 6). Propose data structures, and a locking strategy which allows efficient, parallel and thread-safe computation of the top channels viewed by a given user. Assume that there are multiple goroutines processing the zap events and updating the datastructure and multiple goroutines computing top-3 channels for each user. Provide Go code or pseudocode and explain your design decisions and tradeoffs you consider.

Solution:

Data structures can be similar to the ones used in lab 6 instead the map has ip address as key and another map with top viwed channels as values. The logic to compute the top viewed channel is similar to how the duration was computed in lab 6.

For locking strategy, one simple solution would be to use *fine-grained locking*, with one lock per user ip: Instead of mapping to an int, we could map to another data structure that contains the map or other datastructure containing the top viewed channels and a lock. Whenever multiple goroutines wants to update the top channel for a user ip (the key of the map), the goroutines will only need to obtain the lock on the counter for that particular user. That is, there will be one lock for each user ip. But if there are hundreds of thousands of users this could be a problem since most operating systems do not allow to create so many locks.

A more sophisticated approach could be used to reduce the number of locks that is needed, e.g. the data structure could use the same lock for ip address ranges or with partial matches (first part of the ip for example).

One **critical issue that we must consider** now is that multiple goroutines can add a new user or same existing to the map. If this happens, then one or more of the concurrent addition of the same entry may be lost. This could be solved in two obvious ways:

1. We could add a read-write lock on the top-level map, which allows multiple readers (holding a read-lock on the map) to get the value for a particular ip . These concurrent readers can update the top viewed channels value, but they must hold the lock for that user (lock shared by a group of ips) to do so. To update the map with a new user, the writer must obtain a write-lock on the top-level map before adding a new channel to the map. This has the benefit that new users can be added dynamically, but has the drawback that it needs to use a separate read-write lock for the top-level map.
2. Another alternative could be to pre-initialize the top-level map with all users, and don't allow dynamic updates to the map. This way, we don't need to use a read-write lock on the top-level map since no new users will be added to the map. We will only update the top viewed channel count for the users.

For full 10 points student should at least mention all of the above points and provide a Go code or pseudocode.

A ZapLab Project Description

Introduction

This is the main project in this course. It will take you through some interesting challenges, and hopefully you will be able to use much of the stuff that you have learnt in the previous lab exercises. The project must be written in Go.

Heads up! Before you begin, you should read through the whole document. This will hopefully help you plan your design, so that you can separate code into separate files and make packages and separate structs and so forth that will help you design a good piece of software.

Another Heads up! You are expected to use the provided template code. Remove TODO comments when you have implemented your solution. When submitting, make sure that your program compiles and runs. If have trouble with some of tasks below: (1) Try to get help during the lab exercise hours; (2) If you are unable to get help in time for the deadline, simply comment the non-compiling code, and provide a commit comment describing the problem. This comment should be separate from the first line of the final commit message, which should say: `username lab7 submission`

Collecting Channel Zaps

Imagine that you are working for ZapBox, an Internet and Cable service provider. ZapBox has deployed a huge number of set-top boxes at customer homes that allows them to watch TV over a fiber optic cable. The TV signal is distributed to customers based on a multicast stream for each available TV channel in ZapBox's channel portfolio. Recently, ZapBox commissioned a software update on their deployed set-top boxes. After this software update, the set-top box will send a UDP message to a server every time a user changes the channel on their set-top box. In addition to channel changes, a few other items of interest may also be sent. Thus, a message sent by a set-top box may contain information about either channel changes, volume, mute status, or HDMI status. The content depends on the actions of the different TV viewers. Below is shown a few samples of the message format:

```
2013/07/20, 21:56:13, 252.126.91.56, HDMI_Status: 0
2013/07/20, 21:56:55, 111.229.208.129, MAX, Viasat 4
2013/07/20, 21:57:48, 98.202.244.97, FEM, TVNORGE
2013/07/20, 21:57:44, 12.23.36.158, Canal 9, MAX
2013/07/20, 21:57:46, 81.187.186.219, TV2 Bliss, TV2 Zebra
2013/07/20, 21:57:42, 61.77.4.101, TV2 Film, TV2 Bliss
2013/07/20, 21:57:42, 203.124.29.72, Volume: 50
2013/07/20, 21:57:42, 203.124.29.72, Mute_Status: 0
```

Each line above represents an event, triggered by a single TV viewer's action, either to change the channel on their set-top box, or adjust the volume and so forth. These set-top box events are sent in text format shown above. The fields are separated by comma and have the meaning shown in the table below. Note that the message format with 5 fields represents channel change events, while a message with only 4 fields contains a *status change* in the 4th field, and no 5th field.

	Field No.	Field Name	Description
	1	`Date`	The date that the event was sent.
	2	`Time`	The time that the event was sent.
	3	`IP`	The IPv4 address of the sending set-top box unit.
	4	`FromChan`	The previous channel of the set-top box.
	5	`ToChan`	The new channel of the set-top box.
	4	`StatusChange`	A change in status on the set-top box.

A *StatusChange* may contain one of the following entries:

StatusChange	Value range	Description
`Volume:`	0-100	The volume setting on the set-top box.
`Mute_Status:`	0/1	The mute setting on the set-top box.
`HDMI_Status:`	0/1	The HDMI status of the set-top box indicates whether or not a TV is connected to the set-top box and powered on.

Traffic Generator

For the purposes of this lab project, we have built a traffic generator to simulate the set-top box events generated by ZapBox's customer set-top boxes. The traffic generator resends set-top box events loaded from a large dataset obtained from real traffic. The IP addresses have been scrambled and do not represent a real set-top box. The traffic generator works by synchronizing the timestamp obtained from the dataset with the local clock on the simulator machine. The date is not synchronized.

In a real deployment, the traffic would typically be sent from set-top boxes using UDP and received at a single UDP server, where the data can be processed. However, to make the simulator scale to multiple receiver groups (you the students), we have instead set up the traffic generator on a single machine multicasting each set-top box event to a single multicast address.

Part 1: Building a Zap Event Processing Server

The objective of this part is to develop a UDP multicast server that will process the events that are sent by the set-top box clients (in our case the traffic generator). The server can run on one of the machines in the Linux lab. Your server should be able to receive UDP packets from the traffic generator using multicast address and port:

224.0.1.130:10000

Note that since the traffic generator is continuously sending out a stream of zap events, it may be difficult to work with this part of the lab on your own machine. The multicast stream is only available on the subnet of the Linux lab. It is therefore recommended that you work on the lab machines, either physically or remotely using ssh.

Tasks and Questions:

(a) Build a UDP zapserver that listens to the IP multicast address and port number specified above. Your server *must not* echo anything back (respond) to the traffic generator. Your server should only receive zap events in a loop. In this task, the server only needs to print to the console whatever it receives from the traffic generator. Hint: `net.ListenMulticastUDP`.

(b) Develop a data structure for storing individual zap events. The struct must contain all the necessary fields to store channel changes (ignore storing status changes for now). The test cases in `chzap_test.go` should pass. The main task here is to implement the constructor `NewSTBEvent()` which can be used by your server when it receives a zap event. In addition the struct should have the following methods. See the template in `chzap.go`.

Method	**Description**
`NewSTBEvent()`	Returns one of three items depending on the input string, either a channel zap event, a status change event, or an error.
`String() string`	Return a string representation of your struct.
`Duration(provided ChZap) time.Duration`	Return the duration between two zap events: the receiving *zap* event and the *provided* event.

Hints: `time.Time` package, Methods: `time.Parse()`, `strings.Split()`, `strings.TrimSpace()`, Layout: `const timeLayout = "2006/01/02, 15:04:05"`

(c) The next task is to use `zlog/simplelogger.go` (available on github) to store the channel changes received on your zapserver.

1. Use the API of the simple logger to compute the number of viewers on `NRK1` periodically, once every second. Print the output to the console.
2. Implement the same for `TV2 Norge`. They should both be printed to the console on a separate line. Measure the time it takes to compute the `Viewers()` function using `TimeElapsed()`.

Optional task: In the tasks above, you may observe different outputs at different times of the day, reflecting the actual number of viewers that were actively changing channels at your current time of day (on a previous date). Study the output at different times of the day, perhaps coinciding with well-known TV programs on the two channels in question. Document and explain what you observe. Is there any correlation between the data for NRK1 and TV2?

(d) Take note of the measurements obtained for the `Viewers()` function over time. What does these results show? What could be the cause of the observed problem?

(e) Implement a function that can compute a *list of the top-10* channels. Call this function periodically, once every second. Hint: Results returned from the `ChannelViewers()` method defined in the `ZapLogger` interface can be sorted.

Note that the underlying data structure used so far precludes an efficient implementation.

(f) Implement a new data structure that avoids the problems that you should have identified with the simple slice-based storage solution. Implement the data structure so that it can support you with keeping track of the top-10 list of channels. Your implementation must adhere to the `ZapLogger` interface. Hint: You do not need to store all the zap events to compute the number of viewers for each channel.

Part 2: Publish/Subscribe RPC Client and Server

(a) Associated with your UDP zapserver, implement an RPC-based server (called a publisher) that takes `Subscribe()` requests from external clients wishing to subscribe to a stream of viewership statistics (the top-10 list). A subscriber client should include the following information in its request:

```
type Subscription struct {
        ClientAddr  string
        RefreshRate int
}
```

Note that, in order for the publisher to send publications (viewership statistics) to its subscribers (the clients), also the clients must implement an RPC server running on the provided `ClientAddr`.

- The `ClientAddr` specifies the IP address and port number of the subscriber client.
- The `RefreshRate` specific how often a subscriber wishes to be notified.
- The RPC server is serving statistics based on zap events from the zap storage, while continuously updating the server's storage (the state of the server).
- The RPC server and the zapserver part receiving zap events should be implemented as separate goroutines, preferably in separate files.
- Assume that the refresh rate is one second or more.

Implement the publisher RPC server and the corresponding subscriber RPC client. The client should display the viewership updates as they are received from the RPC server, leaving the refresh rate handling to the server.

1. How would you characterize the access pattern to the server's state?
2. With this access pattern (workload) in mind. How would you protect the server's state to avoid returning a statistics computation that is incorrect or otherwise malformed?

(b) Now we want to analyze the duration between channel change clicks. To do that, we need to store the previous zap event for each IP, so that you can use the `Duration()` method that you developed earlier. You will need to extend your new data structure or add another data structure for storing these durations. Also, extend the `Subscription` struct with an additional field to select the type of statistics the subscription refers to, either viewership or duration statistics. Whatever statistics is chosen, your publisher sends publications to subscribers at the specified refresh rate.

(c) **Extra:** Profile the data structure implemented in Part 2 (a). Implement a data structure that better supports the workload experienced by the zapserver. Profile the new data structure and compare the results to the one implemented in (a).

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

Package files

cond.go mutex.go once.go pool.go race0.go runtime.go rwmutex.go waitgroup.go

type Cond

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

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 can be created as part of other structures. A Cond must not be copied after first use.

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

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

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

type Mutex

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

A Mutex is a mutual exclusion lock. Mutexes can be created as part of other structures; the zero value for a Mutex is an unlocked mutex.

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

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

Once is an object that will perform exactly one action.

▹ 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.

type Pool

```
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
}
```

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.

func (*Pool) Get

```
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

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

Put adds x to the pool.

type RWMutex

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

An RWMutex is a reader/writer mutual exclusion lock. The lock can be held by an arbitrary number of readers or a single writer. RWMutexes can be created as part of other structures; the zero value for a RWMutex is an unlocked mutex.

func (*RWMutex) Lock

```
func (rw *RWMutex) Lock()
```

Lock locks rw for writing. If the lock is already locked for reading or writing, Lock blocks until the lock is available. To ensure that the lock eventually becomes available, a blocked Lock call excludes new readers from acquiring the lock.

func (*RWMutex) RLock

```
func (rw *RWMutex) RLock()
```

RLock locks rw for reading.

func (*RWMutex) RLocker

```
func (rw *RWMutex) RLocker() Locker
```

RLocker returns a Locker interface that implements the Lock and Unlock methods by calling rw.RLock and rw.RUnlock.

func (*RWMutex) RUnlock

```
func (rw *RWMutex) RUnlock()
```

RUnlock undoes a single RLock call; it does not affect other simultaneous readers. It is a run-time error if rw is not locked for reading on entry to RUnlock.

func (*RWMutex) Unlock

```
func (rw *RWMutex) Unlock()
```

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

As with Mutexes, a locked RWMutex is not associated with a particular goroutine. One

goroutine may RLock (Lock) an RWMutex and then arrange for another goroutine to RUnlock (Unlock) it.

type WaitGroup

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

A WaitGroup waits for a collection of goroutines to finish. The main goroutine calls Add to set the number of goroutines to wait for. Then each of the goroutines runs and calls Done when finished. At the same time, Wait can be used to block until all goroutines have finished.

▹ Example

func (*WaitGroup) Add

```
func (wg *WaitGroup) Add(delta int)
```

Add adds delta, which may be negative, to the WaitGroup counter. If the counter becomes zero, all goroutines blocked on Wait are released. If the counter goes negative, Add panics.

Note that calls with positive delta must happen before the call to Wait, or else Wait may wait for too small a group. Typically this means the calls to Add should execute before the statement creating the goroutine or other event to be waited for. See the WaitGroup example.

func (*WaitGroup) Done

```
func (wg *WaitGroup) Done()
```

Done decrements the WaitGroup counter.

func (*WaitGroup) Wait

```
func (wg *WaitGroup) Wait()
```

Wait blocks until the WaitGroup counter is zero.

Subdirectories

Name	Synopsis
..	
atomic	Package atomic provides low-level atomic memory primitives useful for implementing synchronization algorithms.