

NOTATER UKE 11 –

Systemkall, Scheduling og vaffelrøre

Det kan være vanskelig for en CPU i noen prosesser å finne ut hvor de kan avbryte en prosess for å så starte på en annen, som f eks kjøring av en Fibonacci kode:

Maskininstruksjoner som regner ut Fibonacci-rekken 1 1 2 3 4 5 13
21 34 55 etc:

1. mov 1, %ax # %ax = 1
2. mov 1, %bx # %bx = 1
3. add %ax, %bx # %bx = %bx + %ax
4. add %bx, %ax # %ax = %ax + %bx
5. jmp 3

La oss si vi har summasjonstegnet som skal summere alle tall fra 1 til 2000. Da kan koden fordeles slik at den ene CPU-en regner ut summen fra 1-1000 og den andre CPU-en regner ut summen fra 1000-2000.

$$S = 1 + 2 + 3 + 4 + \dots + 2000 = \sum_{i=1}^{2000} i$$

- En CPU kan regne ut $\sum_{i=1}^{1000} i$
- En annen CPU kan regne ut $\sum_{i=1001}^{2000} i$

Operativsystemet ser bare maskininstruksjoner og aner ikke hva som foregår.

Programmereren må selv sørge for parallelisering. Tråder (threads) kan kjøre uavhengig på hver sin CPU, men programmereren må fordele jobben på trådene.

Med multicore CPU så deles samme minne, og det er derfor veldig viktig at to prosesser (tasks) ikke ødelegger for hverandre ved å for eksempel skrive til samme minne, kapre for mye CPU-tid eller får systemet til å henge.

Beste løsningen til det er: preemptive multitasking (som betyr at operativsystemet har all makten). Operativsystemer deler ut rettigheter til prosesser, og mengde ram til hver av dem.

PROSESSOR MODUS

Operativsystemet får hjelp fra hardware, og av de viktigste delene med den hjelpen er «prosessor modus».

Alle moderne prosessorer har en **modusbit** som kan begrense hva som er lov å gjøre.

Modusbit switcher mellom bruker og privilegert modus. Det kalles også protection hardware og er nødvendig for å kunne kjøre multitasking. User og kernel mode refererer til to forskjellige moduser for å utføre instruksjoner på en datamaskin. Hvis vi ikke hadde forskjell på user og kernel mode, så vil da i prinsippet enhver prosess kunne ta over kontroll på systemet. Da vil alt avhenge av at den prosessen gjør ting riktig.

- **Brukermodus** (user mode): begrenset aksess av minne og instruksjoner, må beskjæftige seg om tjenester. Modus der en applikasjon eller et program kjører i en datamaskin og bare har tilgang til en begrenset del av datamaskinressursene. Denne modusen brukes for å beskytte systemressursene fra å bli ødelagt eller endret av applikasjoner for å sikre stabiliteten til os. I user mode kan en applikasjon kun utføre enkelte operasjoner, og hvis den prøver å utføre en operasjon som krever høyere tillatelser, vil os avvise det.
- **Privilegertmodus** (kernel mode): alle instruksjoner kan utføres. Alt minne og alle registre kan aksesseres. Modus der os har full tilgang til datamaskinens ressurser. Dette inkluderer tilgang til maskinvaren og ressursene som er nødvendige for å utføre systemoppgaver som styring av minne, håndtering av enheter og administrasjon av prosesser. I kernel mode kan os utføre alle typer operasjoner, inkludert de som krever tilgang til maskinvare og datamaskinens ressurser som normalt sett ikke er tilgjengelig i user mode.

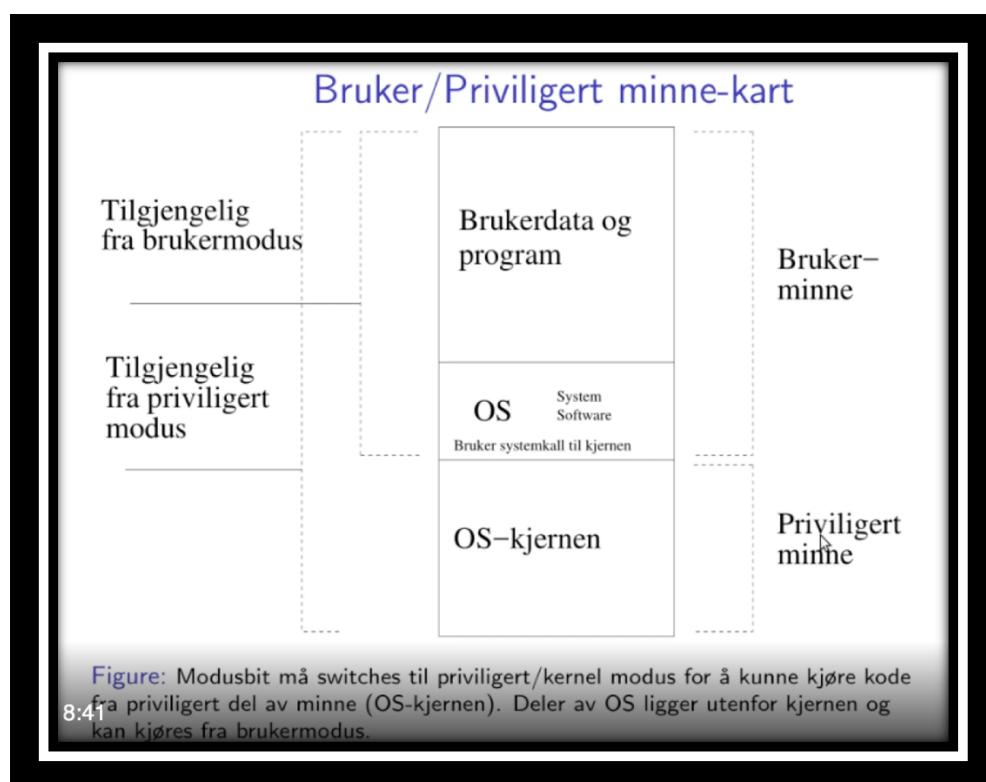
SPØRSMÅL: HVIS DU KJØRER ET SCRIPT SOM ROOT, KJØRES KERNEL MODE?

SVAR: NEI, å kjøre et script som root betyr ikke nødvendigvis at du kjører i kernel mode.

Root-tilgang gir oss en høyere tillatelse til å utføre administrative oppgaver på systemet, men det betyr ikke at man har direkte tilgang til kjernen eller at vi utfører oppgaver i kjernen.

Root tilgang gir oss muligheten til å utføre oppgaver som krever høyere tillatelser enn en vanlig brukerkonto. I de fleste tilfeller vil vi fortsatt kjøre i brukermodus når vi utfører disse oppgavene, med mindre oppgaven krever direkte tilgang til systemressurser som bare er tilgjengelige i kjerneområdet. Men selv da vil vi ikke nødvendigvis utføre oppgaven i kjernen selv, men heller å bruke spesielle systemkall for å få tilgang til kjerneområdet fra brukermodus.

HVORDAN OS EFFEKTIVT KONTROLLERER BRUKERPROSESSER



Over i minnekartet (bilde av RAM) vises hvordan user og kernel mode ser ut som på RAM-nivå. Hardware-timeren brukes i sammenheng med bruker- og kjerne-modus for å bytte mellom de to modusene etter en bestemt tid. Dette er en vanlig måte å gjøre multitasking på, hvor CPU-en skifter mellom forskjellige oppgaver med en bestemt frekvens.

I brukermodus, utfører applikasjonsprogramvaren en rekke operasjoner, inkludert systemkall til operativsystemet. Når systemkallet er gjort, vil operativsystemet ta kontroll over CPU-en og bytte til kjernemodus for å utføre systemkallet. Når systemkallet er fullført, vil operativsystemet returnere kontrollen til brukermodus.

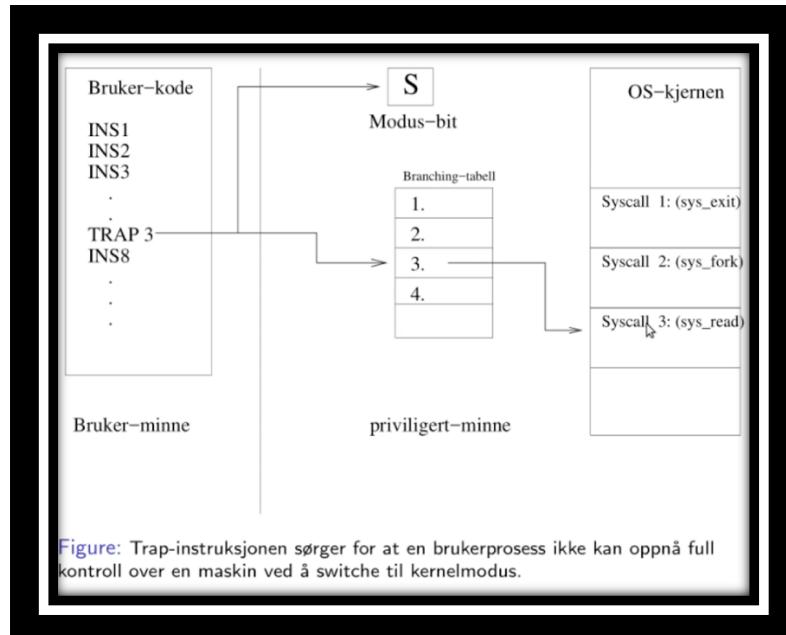
Hardware-timeren brukes for å tvinge operativsystemet til å bytte til kjernemodus etter en bestemt tidsperiode. Dette er en måte å sikre at CPU-en ikke blir bundet opp i en uendelig løkke eller en annen blokkerende operasjon. Når tidsperioden er utløpt, vil hardware-timeren utløse en interrupt (avbrytelse), og CPU-en vil skifte til kjernemodus for å håndtere avbrytelsen. Dette gir operativsystemet muligheten til å utføre andre oppgaver som måtte vente på CPU-ressurser.

Det er mulig å bytte mellom user mode og kernel mode i en datamaskin. Når en prosess utfører en oppgave i user mode og trenger å utføre en oppgave som krever kjerneprivilegier, vil den utløse en trap eller et system call, som vil bytte den fra user mode kernel mode. Kernel mode gir tilgang til ressurser som krever høye privilegier, for eksempel enheter, drivere og andre systemressurser. Når oppgaven er fullført i kernel mode, kan prosessen bytte tilbake til user mode.

Eksempel på når brukerprogram må gjøre noe som ikke kan gjøres fra user mode er å lese fra disk. Operativsystemet har full tilgang og styr på disken. Det er her **systemkall** kommer inn. Systemkall er bare et API mot operativsystemkjernen. Det er et sett med operasjoner som et vanlig brukerprogram fra user mode kan benytte til å gjøre.

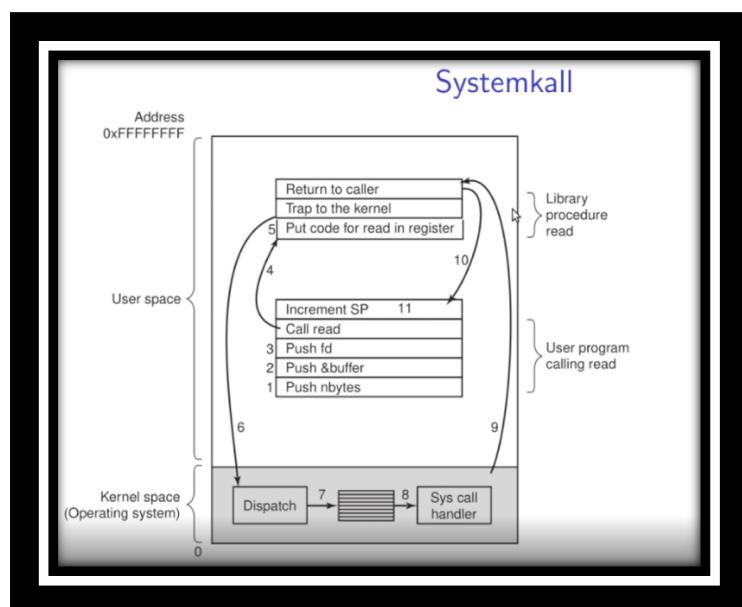
TRAP

En «trap» i programmering refererer til en hendelse eller et signal som kan avbryte kjøringen av en kode eller programvare. Traps brukes vanligvis til feilhåndtering og debugging, og kan brukes til å fange opp feil og avvik som oppstår under kjøring av programvaren. I tillegg kan traps brukes til å håndtere spesielle situasjoner som krever at programvaren stopper eller utfører spesielle handlinger. I operativsystemer brukes traps også til å benytte mellom user mode og kernel mode. Trap switcher til kernel mode og hopper til kode for systemkall i en operasjon.



Til venstre ser vi brukerminne (minne til user mode) der alle instruksjoner foregår også kommer vi til instruksjon 3 der et systemkall skjer. Helt til høyre ser vi Linux-API-et hvor det finnes veldig mange systemkall, hvor enkelte gjøres mer ofte enn andre. Den tredje «`sys_read`» er det å lese fra disk, noe man ikke kan gjøre i user mode.

Trap bytter modusbit til kernel mode SAMTIDIG som den hopper inn i branching tabellen og hopper til første kodelinje i det systemkallet. Første modusbit legges inn i instruksjonsregisteret, da er det neste instruksjon som skal gjøres. En branching tabell er bare en tabell som peker til de ulike systemkallene. Man ser nummer tre i tabellen peker til adressen der i RAM som første instruksjonen etter systemkallet ligger.



Under er det eksempler på noen systemkall

Process management	
Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management	
Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management	
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous	
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

Den første fork for eksempel brukes i Linux for å sette i gang en ny prosess, noe et vanlig brukerprogram ikke kan gjøre, men os må inn.

Linux og Windows systemkall		
UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve (none)	CreateProcess = fork + execve	
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link (none)	Win32 does not support links	
unlink	DeleteFile	Destroy an existing file
mount (none)	Win32 does not support mount	
umount (none)	Win32 does not support mount	
chdir	SetCurrentDirectory	Change the current working directory
chmod (none)	Win32 does not support security (although NT does)	
kill (none) ↗	Win32 does not support signals	
time	GetLocalTime	Get the current time

LINUX-SCEDULING OG PRIORITET

Linux scheduling og prioritering handler om hvordan Linux-kjernen håndterer kjøringen av forskjellige prosesser på systemet. Kjernen har en oppgaveplanlegger som bestemmer hvilken prosess som skal kjøre neste gang en CPU-kjerne blir ledig. Planleggeren tar hensyn til faktorer som prosessens prioritet, antall tidslots en prosess har hatt, og om en prosess har blokkert I/O eller venter på en ressurs.

Prioritetene for en prosess er satt fra 0 til 139, hvor 0 er den høyeste prioriteringen og 139 er den laveste. Normalt sett vil en prosess med høyere prioritet bli gitt CPU-tid før en prosess med lavere prioritet. Prioriteten kan endres dynamisk av brukeren eller programvaren, men enkelte restriksjoner kan hindre at en prosess øker prioritet uten å ha tilstrekkelige rettigheter.

Linux-kjernen støtter også ulike scheduler-algoritmer, som hver har forskjellige fordeler og ulemper avhengig av bruksområdet. Noen av de vanlige algoritmene inkluderer CFS (Completely Fair Scheduler), som sikrer at hver prosess får en "fair share" av CPU-tiden, og Real-time Scheduler, som gir garantier for å møte strenge tidskrav til sanntidsapplikasjoner.

Generelt sett håndterer Linux-kjernen scheduling og prioritering på en effektiv måte som sikrer at ressursene blir optimalt utnyttet og at forskjellige prosesser kjører uten å forstyrre hverandre.

READY LIST

- Linux 2.6 kjernen deler dynamisk prosessene inn i 140 forkjellige prioritetsklasser.
- Hver prioritetsklasse tilsvarer et antall tildelte ticks i starten av en epoke.
- Hver gang scheduleren kalles, velges prosessen som har høyest prioritet.
- Kjører til den har brukt opp alle sine tildelte ticks, eller gir fra seg CPUen.
- Deretter kalles scheduleren på nytt.

I konteksten av Linux scheduling, refererer "ready list" til listen over prosesser som venter på å bli utført av CPU-en. Når en prosess blir klar for utførelse og det er CPU-tid tilgjengelig, blir den satt på "ready list". Deretter blir prosessene på "ready list" prioritert og valgt av scheduleren for å bli utført basert på deres prioritet og andre faktorer. Så, "ready list" spiller en viktig rolle i planleggingen av prosessene og gir en mekanisme for å administrere og prioritere oppgaver i en Linux-basert datamaskin.

PRIORITET

Prioritetsplanlegging i operativsystemet referer til en metode for å bestemme rekkefølgen der prosessene kjører på en datamaskin. Hver prosess tildeles en prioritet basert på viktigheten og ressursbehovet, og operativsystemet bruker denne prioriteten til å avgjøre hvilken prosess som skal utføres neste.

Vanligvis kan prioriteten for en prosess bli angitt som en numerisk verdi, ofte mellom 0 og 127, hvor høyere tall indikerer høyere prioritet. Operativsystemet gir høyere prioritet til prosesser som krever mer CPU-tid eller som er kritiske for systemets funksjonalitet.

Prioritetsskjemaet kan implementeres på forskjellige måter avhengig av operativsystemet. Noen operativsystemer bruker en forhåndsbestemt tildelingsalgoritme, mens andre operativsystemer kan justere prioriteter dynamisk basert på systembelastning eller andre faktorer.

En **ulempe** med prioritetsskjemaet er at det kan føre til at visse prosesser får for mye CPU-tid mens andre blir oversett. Dette kan føre til at noen applikasjoner kjører tregt eller ikke svarer, mens andre applikasjoner fungerer jevnt. Derfor er det viktig å finne riktig balanse mellom prioritering og rettferdig CPU-allokering.

LINUX 2.6 KJERNEN HAR DYNAMISK PRIORITERINGSALGORITME

Som deler prosessene inn i en rekke prioritetsklasser (også kjent som scheduling classes) basert på egenskapene til prosessen og systemets tilstand. Disse inkluderer:

1. real-time klasser: disse er begrenser på applikasjoner som krever sanntidsrespons, for eksempel audio-og videobehandling
2. round-robin klassen: denne klassen brukes til å planlegge interaktive prosesser og tildeler like mye CPU-tid til hver prosess.

3. kjør-kø klassen: denne klassen brukes til å planlegge lande kjøretidsprosesser
4. batch klassen: denne klassen brukes til å planlegge bakgrunnsprosesser som ikke har noen spesielle krav til responstid

NEED RESCHED

- Hvis det i løpet av epoken kommer et interrupt, vil et flagg `need_resched` bli satt
- Scheduler vil på grunn av dette kjøres etter neste timer-tick.
- Interaktive prosessen gis dynamisk høy prioritet og får dermed bedre responstid ↴
- Ellers måtte de vente helt til en kjørende CPU-intensiv prosess var ferdig med alle sine ticks i en epoke

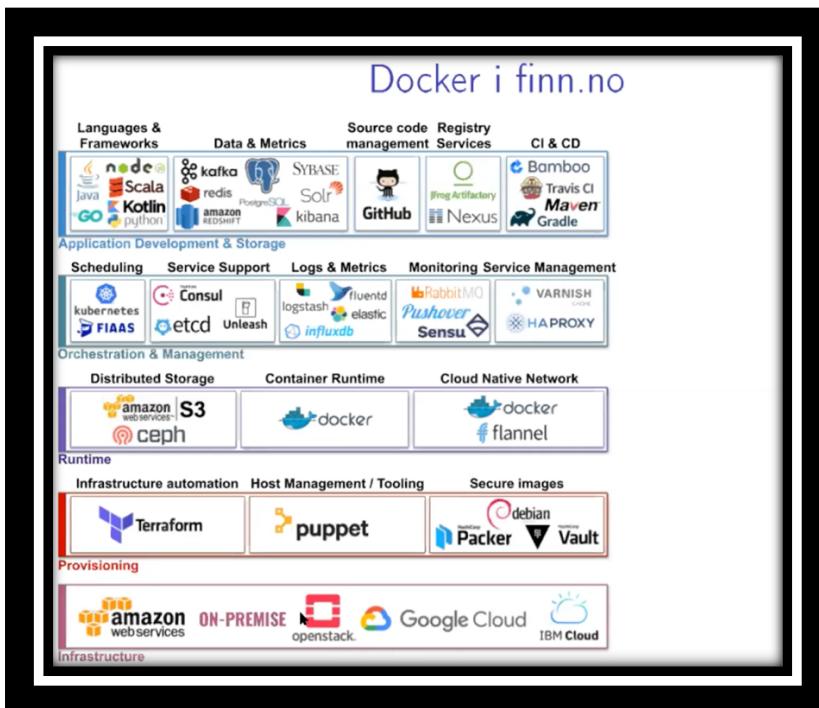
I Linux-kjernen blir flagget «`need_resched`» satt hvis det oppstår et interrupt mens en prosess er i kjøring. Dette indikerer at kjernen må planlegge omfordeling av CPU-ressurser for å gi den nye oppgaven en sjanse til å kjøre umiddelbart. Ved neste mulighet vil kjernen planlegge omfordeling av CPU-tiden for å sikre at alle oppgavene får en rettferdig andel av CPU-ressursene. Dette er en mekanisme som sikrer effektiv multitasking og ressursstyring i operativsystemet.

Når flagget «`need_resched`» settes på grunn av et interrupt i en epoke, vil scheduleren starte på nytt etter timer-tick. Dette gir muligheten til å gi interaktive prosesser, som trenger rask responstid, en høyere prioritet og dermed få en raskere respons fra systemet. Uten denne mekanismen ville interaktive prosesser måtte vente til CPU-intensive prosesser var ferdige med sin tid, noe som kunne føre til lang ventetid og en mindre responsiv brukeropplevelse.

LINUX

Hvorfor docker?

- Enkelt å sette opp og kopiere et helt driftsmiljø
- Bruker mindre ressurser og er raskere å starte og stoppe enn virtuelle maskiner (VM-er)
- Svært viktig del av continuous delivery og continuous development
- Kan brukes til å sette opp store komplekse systemer med Kubernetes (K8s)
 - o Kubernetes er et open-source system for automatisering av distribusjon, skalering og administrasjon av containeriserte applikasjoner. Det er et verktøy som gjør det enklere å administrere og deployere applikasjoner i et distribuert miljø. K8s kan kjøres på en rekke plattformer, inkludert både lokale datamaskiner og skybaserte tjenester.



Kommandoen «df -h» viser informasjon om filsystemet og diskplass. Den viser plasseringen av filsystemet, størrelsen på partisjonen, hvor mye som er brukt, og hvor mye som er ledig. «-h» opsjonen betyr «human readable» som viser informasjonen i en lesbar form for mennesker.

```
root@osG70:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
udev            476M   0M  476M  0% /dev
tmpfs           98M   1M  87M  11% /run
/dev/xvda1      8.2G  6.7G  1.2G  86% /
tmpfs           486M   0M  486M  0% /run/lock
tmpfs           5.0M   0M  5.0M  0% /run/cgroup
overlay         8.2G  6.7G  1.2G  86% /var/lib/docker/overlay2/974a89cedcc49a645102f08707bd0929f94cbe653099481d5f0835d72918617/merged
shm             64M   0M  64M  0% /var/lib/docker/containers/e1727a4988a80dc10bc1638d11b7aa8698af6ecd17a5049df8fb3d50c0c9ac6/mounts/shm
overlay         8.2G  6.7G  1.2G  86% /var/lib/docker/overlay2/0ca5b532c71756563ec726e029327cbc6335e5b1469ab872b2670889df54aa8c/merged
shm             64M   0M  64M  0% /var/lib/docker/containers/9a7e98cb3eb27b65857d04e33971ee2423c5766872cac77035842a5194ec4/mounts/shm
overlay         8.2G  6.7G  1.2G  86% /var/lib/docker/overlay2/f7196216618df69758bfaf2f23d836893141811e058303c724027d3f6293fb352/merged
shm             64M   0M  64M  0% /var/lib/docker/containers/29e4be744ec2f15e799e81350f622ca1b8f4017f81cf240d0ff91211c5bfc8d3/mounts/shm
overlay         8.2G  6.7G  1.2G  86% /var/lib/docker/overlay2/0dc686b744e91c823df9969b4eccb535b958c7691a832a4ed21a2ab975c34ae0/merged
shm             64M   0M  64M  0% /var/lib/docker/containers/2cc04891b19c7edf73f7fe2069c6bd2161714766116d8bb23d138820bb32e50e/mounts/shm
overlay         8.2G  6.7G  1.2G  86% /var/lib/docker/overlay2/e041740609d3199ab4861b5095297424f79963a3e7dc7d429995ddb5c8cccd82a/merged
shm             64M   0M  64M  0% /var/lib/docker/containers/66395137f74dbda8b0bf19a829bae5d75c29629ff6241fabaf6221714bb77a7ab/mounts/shm
overlay         8.2G  6.7G  1.2G  86% /var/lib/docker/overlay2/46b4afb3563a1d6d614d57ec8a2aadce30ea722ce2cb49f71a685403152a8eac/merged
shm             64M   0M  64M  0% /var/lib/docker/containers/0265b152abc22162b3e1078ea1736d71a687201a40689e36953e8296b120/mounts/shm
tmpfs           98M   0M  98M  0% /run/user/1001
root@osG70:~#
```

Images-ene tar litt plass, men ikke så voldsomt. Hårek har under 6 containere som kjører, altså 6 fullverdige operativsystemer, i dette tilfellet kjører de webservere. Men de bruker veldig lite diskplass og relativt lite minne også.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
10265b152acb	haugerud/jedubuntu:1.2	"bash"	11 hours ago	Up 11 hours	80/tcp	elegant_ptolemy
5639513774d	web	"/usr/sbin/apachectl..."	11 hours ago	Up 11 hours	0.0.0.0:8086->80/tcp	webindex6
2cc04891b19c	web	"/usr/sbin/apachectl..."	11 hours ago	Up 11 hours	0.0.0.0:8085->80/tcp	webIndex
29e4be744ec2	cb218c4c37e9	"/usr/sbin/apachectl..."	11 hours ago	Up 11 hours	0.0.0.0:8084->80/tcp	webVol4
9a7e98cb3beb	3a6d4be10758	"/usr/sbin/apachectl..."	11 hours ago	Up 11 hours	0.0.0.0:8083->80/tcp	webVol
e1727a4808a8	d498f0646171	"/usr/sbin/apachectl..."	12 hours ago	Up 12 hours	0.0.0.0:8081->80/tcp	webNy

ER CONTAINERE VM-ER?

Containere er ikke det samme som virtuelle maskiner (VM-er). En virtuell maskin emulerer et fysisk system og krever et eget operativsystem og kernel for å kjøre applikasjoner. På en annen side deler containere operativsystemkjernen med vertsmaskinen, og isolerer applikasjonene fra hverandre ved hjelp av namespaces og cgroups.

Containere er derfor lettere og raskere enn virtuelle maskiner. De kan opprettes og startes raskt og tar mindre plass, siden de ikke trenger en egen kopi av operativsystemet. Derfor har containerteknologi blitt populært for å distribuere og skalere applikasjoner i skybaserte miljøer.

Kommandoen «`docker container ps -a`» er en kommando som viser alle docker-containere som kjører eller har kjørt på systemet, inkludert de som har stoppet eller avsluttet. Informasjonen som vises inkluderer navn, ID, status, opprettet dato, portoppføring og hvilket bilde containeren ble opprettet fra.

«docker prune» er en kommando som brukes til å rydde opp i ubrukte Docker-ressurser, som for eksempel containere, bilder og nettverk. Når du kjører **docker prune**, vil Docker fjerne alle ressurser som ikke er i bruk av dine Docker-containere.

```
root@osG70:~# docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
10265b152acb        haugerud/jedubuntu:1.2   "bash"              11 hours ago       Up 11 hours        80/tcp              elegant_ptolemy
66395137f74d        web                 "/usr/sbin/apache2" 11 hours ago       Up 11 hours        0.0.0.0:8086->80/tcp
2cc04891b19c        web                 "/usr/sbin/apache2" 11 hours ago       Up 11 hours        0.0.0.0:8085->80/tcp
29e4be744ec2        cb218c4c37e9      "/usr/sbin/apache2" 11 hours ago       Up 11 hours        0.0.0.0:8084->80/tcp
9a7e98cb3beb        3a6d4be10758    "/usr/sbin/apache2" 11 hours ago       Up 11 hours        0.0.0.0:8083->80/tcp
e1727a4808a8        d498f0646171    "/usr/sbin/apache2" 12 hours ago      Up 12 hours        0.0.0.0:8081->80/tcp
root@osG70:~#
```

«docker container ls -q» vil kun liste id-ene:

```
root@osG70:~# docker container ls -q
10265b152acb
66395137f74d
2cc04891b19c
29e4be744ec2
9a7e98cb3beb
e1727a4808a8
root@osG70:~#
```

Under er et eksempel på hvordan man stopper den aller første containeren:

```
root@osG70:~# docker container stop 10265b152acb
10265b152acb
root@osG70:~#
```

Under er det skrevet et skript som vil stoppe alle containere:

```
root@osG70:~# for c in $(docker container ls -q); do docker container stop $c; done
66395137f74d
2cc04891b19c
29e4be744ec2
9a7e98cb3beb
e1727a4808a8
root@osG70:~#
```

«docker system prune -af» er en kommando som i docker brukes for å fjerne alle stoppete containere fra systemet, uavhengig av status.

```
root@osG70:~# docker system prune -af
Deleted Containers:
10265b152acb22162b3e1078ea1736d711af687201a40689e36953e8296b120
702610c0df5921679bddd701f9893ce4f08aa7f7c77a1e983eea808b6716726b
e1727a4808a80dc10bc1638d11b7aad8698af6ecd17a50448df8b3d50c0c9ac6
210a1a82c0467346d3056d1b60286083df37c8faf4ca8e33072b2544ccceb403
root@osG70:~#
```

Hvis man skriver «docker container ls -a» etter å ha kjørt kommandoen over så dukker det ikke opp noe. Det fjerner også alle image «docker images» eller «docker image ls -a».

Hvis man eksplisitt bare vil fjerne image skriver man:

```
root@osG70:~# docker image prune -af
Total reclaimed space: 0B
```

Når man lister opp diskbruk igjen med «df -h» så har Hårek 1.6 gigabyte.

```
root@osG70:~# df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev            476M   0    476M  0% /dev
tmpfs           98M   11M   87M  11% /run
/dev/xvda1      8,2G  6,3G  1,6G  81% /
tmpfs           486M   0    486M  0% /dev/shm
tmpfs           5,0M   0    5,0M  0% /run/lock
tmpfs           486M   0    486M  0% /sys/fs/cgroup
tmpfs           98M   0    98M  0% /run/user/1001
root@osG70:~#
```

FORSKJELLEN PÅ IMAGE OG CONTAINER

En docker-image er en statisk fil som inneholder alle komponentene som trengs for å kjøre en applikasjon, mens en container er en kjørende instans av en image, med alle konfigurasjonene og tilleggskomponentene som kreves for at applikasjonen skal kjøre.

Starter en interaktiv docker container og er inni den:

```
root@osG70:~# docker container run -it ubuntu bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
423ae2b273f4: Pull complete
de83a2304fa1: Pull complete
f9a83bce3af0: Pull complete
b6b53be908de: Pull complete
Digest: sha256:04d48df82c938587820d7b6006f5071dbbfce7ca01d2814f81857c631d44df
Status: Downloaded newer image for ubuntu:latest
root@662468ddd33d:/#
```

Tar «pwd» (print working directory) for å se at vi er inni et fungerende Linux filesystem. Også ser vi at vi er i 18.04:

```
root@662468ddd33d:/# pwd
/
root@662468ddd33d:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@662468ddd33d:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04.4 LTS"
root@662468ddd33d:/#
```

«exit» vil gå ut av containeren og da er den stoppet, den vil ikke dukke opp i docker container ps, men docker container ps -a. Alternativ måtte å gå ut på er control p control q.

Man kan kopiere id-en til containeren som er stoppet og starte den igjen slik: «docker container start ID».

```
root@osG70:~# docker container ps
CONTAINER ID        IMAGE               COMMAND
root@osG70:~# docker container ps -a
CONTAINER ID        IMAGE               COMMAND
562468ddd33d      ubuntu              "bash"
root@osG70:~# docker container start 662468ddd33d
562468ddd33d
```

Nå kan man se at den kjører igjen:

```
root@osG70:~# docker container start 662468ddd33d
562468ddd33d
root@osG70:~# docker container ps
CONTAINER ID        IMAGE               COMMAND
562468ddd33d      ubuntu              "bash"
root@osG70:~#
```

Slik kan man søke etter gamle kommandoer med grep:

```
root@osG70:~# history | grep exec
 61 docker container exec -it f556566d27a2 bash
 69 docker container exec -it 2ad42bba6955 bash
 75 docker container exec -it 2ad42bba6955 bash
196 docker container exec ps 42
197 docker container exec 42 ps
198 docker container exec 42 /bin/ps
199 docker container exec 42 /sbin/ps
201 docker container exec 42 /sbin/ps
202 docker container exec 42 /bin/ps
```

Man kan koble seg til igjen med exec:

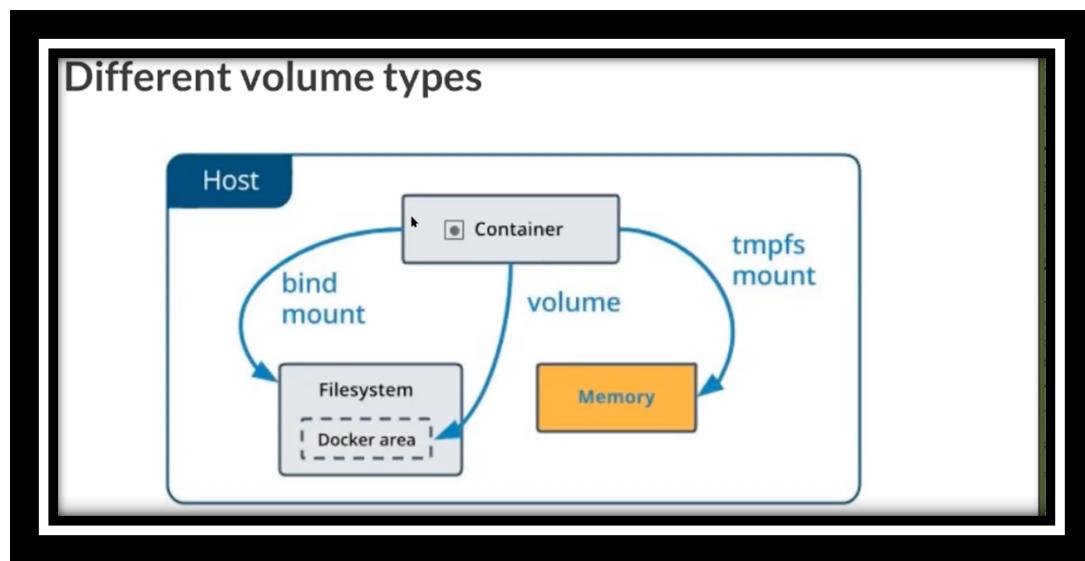
```
root@osG70:~# history | grep exec
root@osG70:~# docker container exec -it 662468ddd33d bash
root@662468ddd33d:/#
```

DOCKER VOLUME

I docker er en «volume» en måte å persistere (lagre data permanent på en lagringsenhet, som en harddisk eller en database) data mellom docker-containere og til og med mellom docker-host og container. En docker-volume kan brukes til å dele data mellom flere containere eller til å sikre at dataene overlever når en container blir ødelagt eller erstattet av en ny.

Det er et par måter å gjøre dette på som bildet under viser:

1. bind mount: den kobler direkte, som en link, man kan ha fil eller mappe her i containeren som peker direkte på en mappe på filsystemet på hosten som kjører containeren. Da kan du endre på filene i hosten, og da endres de i containeren.
2. volume: litt mer abstrakt og foretrukket metode. Et volume defineres av docker Daemon. Da er det docker som står og styrer det her. Kan i større grad gjøre sånn at containere kan dele data med hverandre også.
 - a. Docker Daemon er en kjørende prosess som lytter etter docker CLI-kommandoer og administrerer docker-objekter som containere, bilder, nettverk og vuler. Daemon håndterer også kommunikasjon med Docker Hub og andre registre for å laste ned og laste opp Docker-bilder. Den er en kritisk komponent i docker-arkitekturen og må kjøre på hver docker-verd for at docker skal fungere. Daemon kjører vanligvis som en bakgrunnsprosess på vertsoperativsystemet.
3. man kan også direkte koble minne, men det skal **ikke** vi se på.



Bind mount er å direkte binde hosten. Når docker containeren startes legger man til opsjonen -v. Og host/dir er hosten nede i filsystemet (Se bildet over filsystemet) som i vårt tilfelle er Linux-VM. Også er /app filsystemet i containeren. Vi skal senere koble den til var/www/html på containeren slik at vi kan endre innholdet på webserveren mens den kjører.

1. Bind mount er en måte å knytte en katalog fra vertsfilsystemet direkte til en katalog i Docker-containeren. Dette gjør at filene i den knyttede katalogen i containere blir synkronisert med filene i vertsfilsystemet, og endringer som skjer i en katalog vil

påvirke begge systemene. Dette er nyttig når du trenger å dele data mellom containere og verstsfilsystemet, eller når du trenger å bruke eksisterende data fra verstsfilsystemet i containeren.

2. Volumes er en annen måte å håndtere datalagring i Docker-containere på. Volumer er separate fra verstsfilsystemet, og kan opprettes og administreres av Docker. Volumer kan brukes til å dele data mellom containere eller for å sikre at data overlever når containere blir slettet eller oppgradert.
3. Tmpfs er en tredje måte å håndtere filsystemer i Docker-containere på. Dette er et midlertidig filsystem som opprettes i minnet, og som er nyttig når du trenger å lagre midlertidige data, for eksempel hurtigbuffer eller midlertidige filer som ikke trenger å overleve når containeren stoppes. Tmpfs er raskt og effektivt, men krever at du har nok minne til å holde filsystemet i minnet.

Bind mount, volumes and tmpfs

- Bind mount
 - -v /host/dir:/app
- Volume:
 - -v vol_name:/app
- Tmpfs:
 - --tmpfs /app

The diagram illustrates three types of Docker mounts within a host system:

- bind mount:** Represented by a blue curved arrow pointing from a host directory ('/host/dir') to a container directory ('/app').
- volume:** Represented by a blue curved arrow pointing from a host volume ('vol_name') to a container volume ('/app').
- tmpfs mount:** Represented by a blue curved arrow pointing from host memory ('Memory') to a container directory ('/app').

The host system is shown with a blue header labeled "Host". Inside the host, there is a "Container" represented by a grey box, a "Filesystem" represented by a grey box containing a dashed "Docker area", and a "Memory" represented by an orange box. Arrows indicate the flow of data between these components and the corresponding mount points in the container.

DOCKER BUILD AV APACHE WEB SERVER

En docker-fil inneholder i prinsippet alt en docker container trenger.

Her har Hårek en container som er oppe og kjører:

```
root@osG70:~# docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
562468ddd33d        ubuntu              "bash"              36 minutes ago   Up 34 minutes
root@osG70:~# docker container exec -it 562468ddd33d bash
root@562468ddd33d:~# exit
root@osG70:~# docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
562468ddd33d        ubuntu              "bash"              37 minutes ago   Up 35 minutes
root@osG70:~#
```

Han lager en mappe «webserver» og går inn i den:

```
root@osG70:~# mkdir webserver
root@osG70:~# cd webserver/
root@osG70:~/webserver#
```

Lager en fil som heter Dockerfile

```
root@osG70:~/webserver# jed Dockerfile
```

Her inne i fila skriver han at

```
File Edit View Search Terminal Tabs Help
haugerud@rex: /home/hauger... x haugerud@rex: /home/hauger... x root@662468ddd33d: / x hauger...
F10 key ==> File Edit Search Buffers Windows System Help
FROM ubuntu
RUN apt-get -y update
RUN apt-get -y install apache2
COPY index.html /var/www/html
```

Filen «Dockerfile» er en oppskrift for å bygge et Docker image. Docker images er «templates» eller «mal» for å lage Docker-containere.

I denne Dockerfilen bruker vi ubuntu som base image, og deretter utfører vi to kommandoer for å installere Apache webserver og kopiere filen «index.html» fra vår lokale maskin til Docker-containeren.

«**FROM ubuntu**» indikerer at vi vil bruke ubuntu som grunnlag for vårt Docker image.

«**RUN apt-get -y update**» oppdaterer pakkelister i Ubuntu, mens «**RUN apt-get -y install apache2**» installerer Apache webserver.

Til slutt bruker vi "**COPY**" for å kopiere filen "index.html" fra vår lokale maskin til mappen "/var/www/html" i Docker-containeren. Dette vil erstatte den eksisterende "index.html"-filen i mappen, som vil vises når du går til Apache-serveren i Docker-containeren.

Når denne Dockerfilen bygges ved å kjøre kommandoen "docker build", vil det resultere i et Docker image som inneholder Apache-webserveren og vår "index.html"-fil. Dette image kan deretter brukes til å opprette en Docker-container som vil kjøre Apache-webserveren med vår nettside.

Hvis man noensinne er usikker på hvordan en kommando kan brukes kan man legge til opsjonen «--help».

```
root@osG70:~/webserver# docker build --help
```

Her kjører han fila med å si «**docker build**». -t opsjonen legger til navn (tag) og . refererer til gjeldende katalog, som inneholder Dockerfile som ble laget med jed.

```
root@osG70:~/webserver# docker build -t webapache .
Sending build context to Docker daemon 2.048kB
Step 1/4 : FROM ubuntu
--> 72300a873c2c
Step 2/4 : RUN apt-get -y update
--> Running in e4abeb11b1b5
```

Hårek får feil om at kopiering ikke skjer fordi fila ikke finnes engang så han lager den og legger til innhold i fila med echo:

```
root@osG70:~/webserver# ls -l
total 4
-rw-r--r-- 1 root root 96 mars 20 11:24 Dockerfile
root@osG70:~/webserver# cat > index.html
Hilsen fra Dockerfile container!
```

OGSÅ BYGGER HAN IGJEN. Under ser man at etter bygginga så finnes imaget:

```
root@osG70:~/webserver# docker image ls
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
webapache           latest   347609d4347e  23 seconds ago  189MB
ubuntu              latest   72300a873c2c  3 weeks ago   64.2MB
```

Her er han inne i containeren med «docker container run -it webapache bash»:

```
root@osG70:~/webserver# docker container run -it webapache bash
root@6a44b1eaaee7:/#
```

Her er apache nå installert. Det er flere måter å starte apache på

```
root@6a44b1eaaee7:/# /etc/init.d/apache
apache-htcacheclean apache2
```

Går også ann å gå inn og starte den slik som under, også gå ut igjen med control p control q.

Hvis ikke så stopper hele serveren.

```
root@6a44b1eaaee7:/# /etc/init.d/apache2 start
 * Starting Apache httpd web server apache2
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.3. Set the 'ServerName' directive globally
to suppress this message
 *
```

Hvis man lister «docker container ps» kan man se at man har webapache oppe og kjørende:

```
root@osG70:~/webserver# docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
6a44b1eaaee7        webapache          "bash"             About a minute ago   Up About a minute
562468ddd33d        ubuntu              "bash"             About an hour ago    Up About an hour
mystifying_merkle   wonderful_meitner
```

Over kan du se at vi ikke har gitt den noe port. Man kan sjekke med curl:

```
root@osG70:~/webserver# curl localhost
```

Vi må først stoppe containeren:

```
root@osG70:~/webserver# docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
6a44b1eaaee7        webapache          "bash"             About a minute ago   Up About a minute
562468ddd33d        ubuntu              "bash"             About an hour ago    Up About an hour
mystifying_merkle   wonderful_meitner
root@osG70:~/webserver# docker container stop 6a
```

Nå prøver vi å kjøre igjen med operasjonen -p som sier at all trafikk skal videresendes til 80:

```
root@osG70:~/webserver# docker container run -p 8081:80 -it webapache bash
root@2f03735fa1a0:/#
```

Her ser det ikke ut til at webserveren kjører fordi vi stoppet hele containeren

```
root@2f03735fa1a0:/# ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  2.7  0.3  18508  3444 pts/0      Ss  10:33   0:00 bash
root        11  0.0  0.2  34404  2860 pts/0      R+  10:33   0:00 ps aux
```

Starer apache igjen. Filen `/etc/init.d/apache` er en script-fil som brukes til å starte, stoppe og restarte apache webserver på Linux-systemer. Når kommandoen "service apache start" eller "systemctl start apache" kjøres på en Linux-terminal, vil denne filen bli kjørt for å starte Apache. Filen inneholder instruksjoner for å starte Apache-prosessen og konfigurasjonsinformasjon som skal brukes av Apache når det kjører.

Kommandoen `/etc/init.d/apache2 start` starter Apache webserveren på en Linux-basert system som har Apache installert og konfigurert til å kjøre som en systemtjeneste. Dette starter Apache-prosessen i bakgrunnen, og gjør det mulig å betjene websider som er lagret i `/var/www`-mappen.

```
root@2f03735fa1a0:/# /etc/init.d/apache2
apache-htcacheclean apache2
root@2f03735fa1a0:/# /etc/init.d/apache2 start
* Starting Apache httpd web server apache2
AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.17.0.3. Set the 'ServerName' directive globally
to suppress this message
*
```

Nå kan man se port 8081 kan bli sendt til port 80 på containeren.

```
root@osG70:~/webserver# docker container ps
CONTAINER ID        IMAGE       COMMAND       CREATED          STATUS          PORTS          NAMES
2f03735fa1a0        webapache   "bash"        40 seconds ago   Up 36 seconds   0.0.0.0:8081->80/tcp   adoring_bell
662468ddd33d        ubuntu      "bash"        About an hour ago   Up About an hour   wonderful_meitner
```

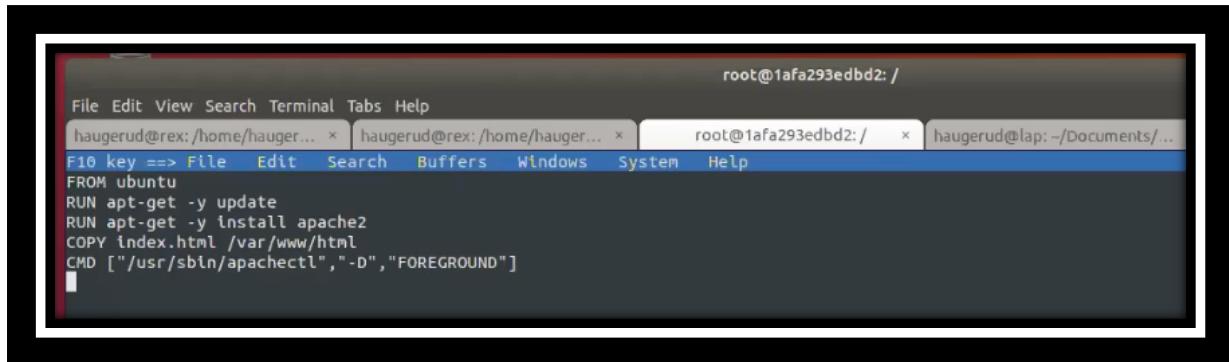
```
root@osG70:~/webserver# curl localhost:8081
Hilsen fra Dockerfile container!
```

Kan sjekke på chrome



ENDRE DOCKERFILE SLIK AT APACHE STARTES NÅR CONTAINEREN STARTER

Endring i jed fila Dockerfile:



Bygge igjen:



Bygge containeren igjen:



SPØRSMÅL: Kan man ha flere Dockerfile i en container

SVAR: Nei, i utgangspunktet har man en slik fil og den bygger en container. Men man kan starte flere instanser av denne containeren.

HVORDAN BYGGE ETT IMAGE OG STARTE MANGE WEBSERVERE PÅ FORSKJELLIGE PORTNUMMER

```
root@osG70:~/webserver# docker container run -p 8081:80 -d webapache
31b11625e922561e3802bec8ca872e481083ed6b93b9610218b1dfcfe2aa86714c
root@osG70:~/webserver# docker container run -p 8082:80 -d webapache
3108af547e98247642ba9c529a0cd21c7488a5a61b04715e4e5569e28d8c667
root@osG70:~/webserver# docker container run -p 8083:80 -d webapache
0d057786ce9e5a3f4890372388f3ae35c9d297fb6f379b9d42156c2de88cb65
root@osG70:~/webserver# docker container ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
0d057786ce9e        webapache          "/usr/sbin/apachectl"   29 seconds ago    Up 25 seconds      0.0.0.0:8083->80/tcp   hungry_taussig
3108af547e98        webapache          "/usr/sbin/apachectl"   37 seconds ago    Up 34 seconds      0.0.0.0:8082->80/tcp   gallant_villain
31b11625e922        webapache          "/usr/sbin/apachectl"   54 seconds ago    Up 51 seconds      0.0.0.0:8081->80/tcp   gracious Aryabhata
```

HVORDAN ENDRE PÅ INNHOLDET FOR EN KJØRENDE DOCKER WEB SERVER

I utgangspunktet må man gå inn i containeren og endre på innholdet i fila med «docker container exec -it ID bash». Bevege seg til «cd /var/www/html» også «cat index.html».

Legge til nytt innholdt slik:

```
root@31b11625e922:/var/www/html# cat >> index.html
Fra 8081
```

Også gå ut med «read escape sequence»

```
root@31b11625e922:/var/www/html# read escape sequence
```

HAR MAN FLERE IMAGE I EN CONTAINER

Nei, en container kan kun kjøre et enkelt Docker-image av gangen. Når du kjører en container, er den isolert fra andre containere og kan bare kjøre prosesser som er spesifiser i det Docker-imaget som brukes til å starte containeren. Hvis man ønsker å kjøre flere prosesser samtidig, kan man vurdere å bruke verktøy som Docker Compose eller Kubernetes for å administrere flere containere som jobber sammen som en helhet.

Vår group VM kjører debian:

```
root@osG70:~/webserver# cat /etc/debian_version
10.3
```

Når vi sier «FROM ubuntu» i skriptet Dockerfile, inneholder ubuntu alt til som skal få debian til å se ut som et ubuntu 18.04. når man da installerer apache kommer ett nytt lag på systemet.

HVORDAN FÅ EN LOKAL FIL PÅ SERVEREN TIL Å VISES AV EN DOCKER WEBSERVER

(siden ble ikke vist pga manglende rettigheter på serveren. Chmod 755 /root gjorde at den ble løst, docker kommandoen for å starte containeren var riktig)

kjøre en Docker-container fra et bilde som heter "webapache"

- opsjonen "-v /root:/var/www/html" settes en bind mount mellom "/root" på vertsooperativsystemet og "/var/www/html" i containeren, slik at endringer i "/root" vil bli reflektert i "/var/www/html" i containeren
- opsjonen "-p 8085:80" vil sette opp en port-forwarding fra port 8085 på vertsooperativsystemet til port 80 i containeren, slik at webserveren kan nås fra en nettleser på vertsooperativsystemet på <http://localhost:8085>
- opsjonen "-d" betyr at containeren skal kjøre i bakgrunnen

```
root@osG70:~/webserver# docker container run -v /root:/var/www/html -p 8085:80 -d webapache  
bc0ca2d6cc6fea15926d8f9e2436b7c90358ab00ee6aca38b89bdcb6d50977ea  
root@osG70:~/webserver# curl localhost:8081
```

Apache2-installasjonen spør nå interaktivt om tidssone og bygningen vil henge. Legg derfor inn riktig tidszone som andre og tredje linje i Dockerfile:

```
ENV TZ=Europe/Oslo  
RUN ln -snf /usr/share/zoneinfo/$TZ /etc/localtime && echo $TZ > /etc/timezone
```