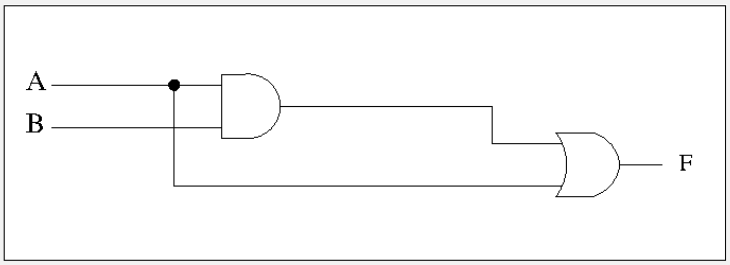
Løsningsforslag eksamen 2022 Operativsystemer

Datamaskinarkitektur

1)

Datamaskinarkitektur

Hva blir det logiske uttrykket F(A,B) for denne kretsen?



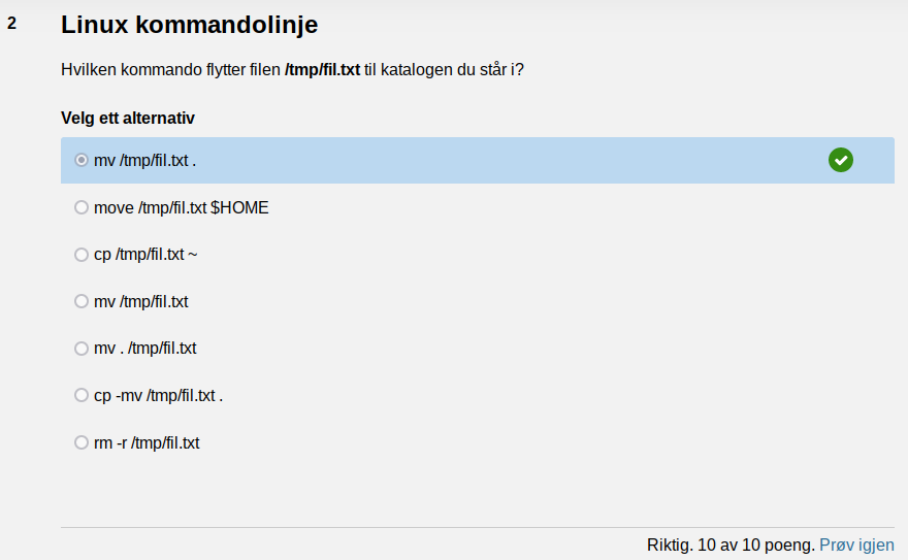
Velg ett alternativ

- $A \cdot B + B$
- $(A + B) \cdot A$
- $\overline{A + B}$
- $A \cdot B + A$ ✓
- $(A + B) \cdot B$
- $A \cdot B \cdot A$
- $\overline{A} \cdot B + \overline{A}$

Riktig. 10 av 10 poeng. Prøv igjen

Linux kommandolinje

2)



3)

Linux i praksis

4)

```
kan@os:~$ cat file1
NNlkD
```

5) Man kan lete manuelt eller bruke en kommando som:

```
kan@os:~$ find . -name file2
./clk/hfi/inv_mpu6050/file2
kan@os:~$ chmod 600 ./clk/hfi/inv_mpu6050/file2
kan@os:~$ cat ./clk/hfi/inv_mpu6050/file2
zFyIh
```

6)

```
kan@os:~$ sudo su
root@os:/home/kan# cat /root/.../...
XTyZa
```

7)

```
root@os:~# docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED      STATUS                    PORTS     NAMES
85ce4157775f   ng        "/docker-entrypoint..."   3 days ago   Exited (0) 3 days ago             nervous_beaver
85ce4157775f
```

8)

```
root@os:~# docker start 85ce4157775f
85ce4157775f
root@os:~# curl localhost
iDNB4
```

9)

```
root@os:~# docker cp 85ce4157775f:/stegosaurus.jpeg .
root@os:~# steghide extract -sf stegosaurus.jpeg
Enter passphrase:
wrote extracted data to "fil.txt".
root@os:~# cat fil.txt
r27XT
```

10) To typiske kjøringer:

```
kan@os:~/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 0
Avslutter; svar verdi: 7237388
Avslutter; svar verdi: 10395395
Main avslutter; svar verdi: 10395395
kan@os:~/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 0
Avslutter; svar verdi: 10072023
Avslutter; svar verdi: 10403016
Main avslutter; svar verdi: 10403016
kan@os:~/lock$
```

I thread.c lager main tråder som kjører uavhengig av hverandre og der begge med funksjonen inc() oppdaterer en felles variabel med navn svar ti millioner ganger. Deretter venter main på at begge trådene skal bli ferdig og skriver ut den totale summen. Her øker begge tråder verdien til variabelen like mange ganger og dermed vet vi at verdien må bli det dobbelte av det hver tråd øker med hvis det ikke inntreffer en race condition; altså 20 millioner. Når programmet kjøres blir resultatet forskjellig hver gang og bare i overkant av 10 millioner. Det betyr at race conditions intreffer hele tiden og at trådene overskriver hverandres resultater. Siden det er to tilgjengelige CPUer vil de kjøre på hver sin CPU og siden det ikke er noen koordinering CPUene imellom om å vente på hverandre når en variabel skal hentes fra RAM, vil de fortløpende overskrive

hverandres resultater. (at svaret er rett over 10 millioner er forenlig med det som skjer hvis begge henter ut verdien 1 omtrent samtidig og øker den til 2 og så skriver tilbake omtrent samtidig, da vil den totale økningen være 1, mens den burde vært 2)

11) Vanligvis skjer dette:

```
kan@os:~/lock$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 2023319
Avslutter; svar verdi: 19379144
Avslutter; svar verdi: 20000000
Main avslutter; svar verdi: 20000000
```

men ca en av fem ganger skjer noe lignende dette:

```
kan@os:~/lock$ taskset -c 0 ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 2275142
Avslutter; svar verdi: 19399210
Avslutter; svar verdi: 16998306
Main avslutter; svar verdi: 16998306
```

Når programmet kjøres med `taskset -c 0` vil begge trådene tvinges til å kjøre på samme CPU. Dermed vil de stort sett kjøres hver for seg og resultatet blir 20 millioner. Men en sjelden gang kommer det en context switch rett etter at en tråd har økt et rigister lokalt med en og før tråden har rukket å skrive til RAM og da vil den andre trådens resultater bli overskrevet når en context switch går tilbake til den første tråden igjen. Dette må også bety at den ene linjen `svar++` i programmet utgjør flere linjer maskinkode, det vil si at verdien av svar først lastes ned i et register, så økes og lastes opp til RAM igjen (se neste oppgave).

12) Når man kompilerer en.c på denn måten får man som resultat en fil en.s som viser assembly-kode som tilsvarer den maskinkoden som lages når en.c blir kompilert til maskinkode. Om man ser på koden inne i `enlinje:` som inneholder den kompilerte assemblykoden for C-instruksjonen `svar++`, kan man se følgende tre linjer:

```
movl svar(%rip), %eax
addl $1, %eax
movl %eax, svar(%rip)
```

Disse linjene viser at svar først blir lastet inn fra RAM til registeret %eax, dette registeret økes med en og resultatet lastes ut i RAM igjen. Hvis det skjer en context switch til den andre tråden etter den første eller den andre av disse instruksjonene, vil tredje linje overskrive alle addisjonene som den andre tråden gjør i mellomtiden. Og dette forklarer resultatene fra forrige oppgave.

13) Når programmet kompileres med minimal.s vil økningen av variabelen svar utføres ved hjelp av en enkelt instruksjon incl `svar(%rip)`. Dermed vil økningen av variabelen svar oppdateres i RAM uansett når en context switch måtte intreffe, for instruksjonen fullføres før context switchen utføres. Og da vil de to trådene ikke ødelegge for hverandre, så lenge de kjører på samme CPU-kjerne og det sørger taskset for.

14) To typiske kjøringer kan nå se slik ut:

```
kan@os:~/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 0
Avslutter; svar verdi: 9717279
```

```
Avslutter; svar verdi: 10465040
Main avslutter; svar verdi: 10465040
kan@os:~/lock$ ./a.out
Starter; svar verdi: 0
Starter; svar verdi: 0
Avslutter; svar verdi: 9969943
Avslutter; svar verdi: 10177571
Main avslutter; svar verdi: 10177571
```

Igjen ser vi at oppdateringen av variabelen svar i RAM ikke blir synkronisert når to forskjellige CPUer henter og oppdaterer den samme variabelen og de overskriver hverandres økninger av variabelen. Dette til tross for at oppdateringen utføres av en enkelt instruksjon `incl svar(%rip)`. Likevel må verdien hentes inn til CPUen via minnebussen, økes med en og sendes tilbake igjen. Og det er ingen kontroll på om den andre CPUen henter inn verdien av variabelen svar samtidig. Begge vil da samtidig øke verdien med en og skrive tilbake til RAM. Igjen forsvinner nesten halvparten av oppdateringene.

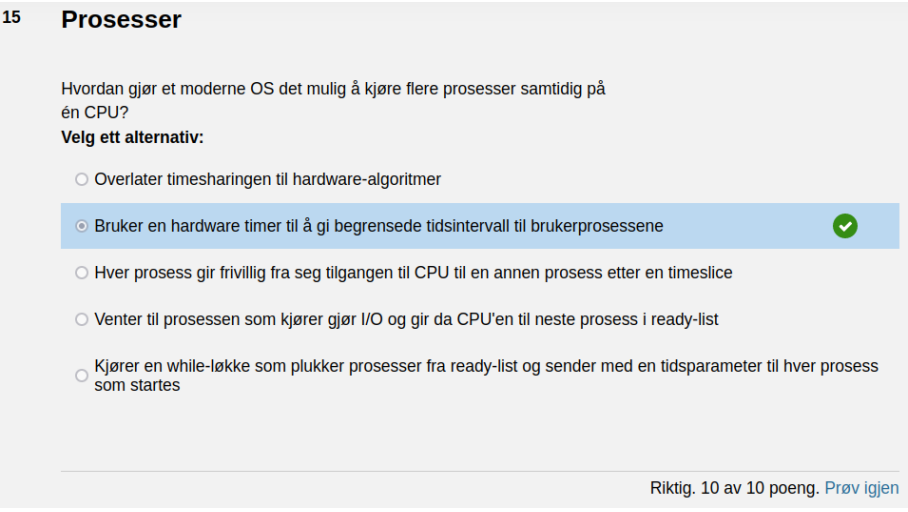
Man kan serialisere trådene ved å utføre assembly-instruksjonen lock før incl:

```
.globl  enlinje
enlinje:
        lock
        incl svar(%rip)
        ret
```

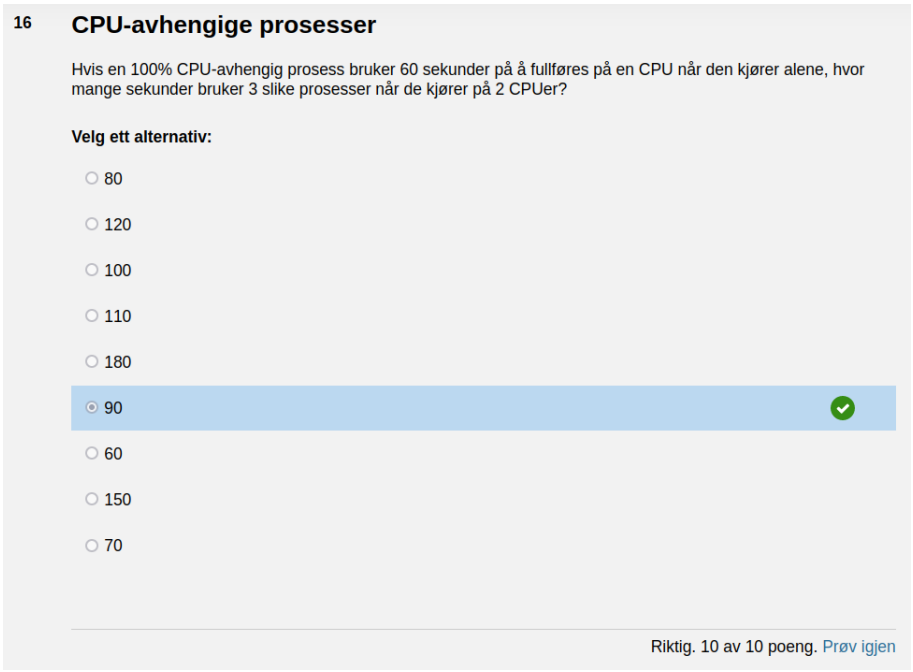
Den vil for neste instruksjon som utføres, incl, minnebussen låses slik at instruksjoner på andre CPUer ikke samtidig kan hente eller lagre noe i RAM. Dette sikrer at instruksjonen etter lock som utføres på en variabel i minne får avsluttet hele sin operasjon uten at RAM endres. Dermed blir svaret hver eneste gang 20 millioner. (Og det tar lengre tid å kjøre programmet pga synkroniseringen)

Prosesser

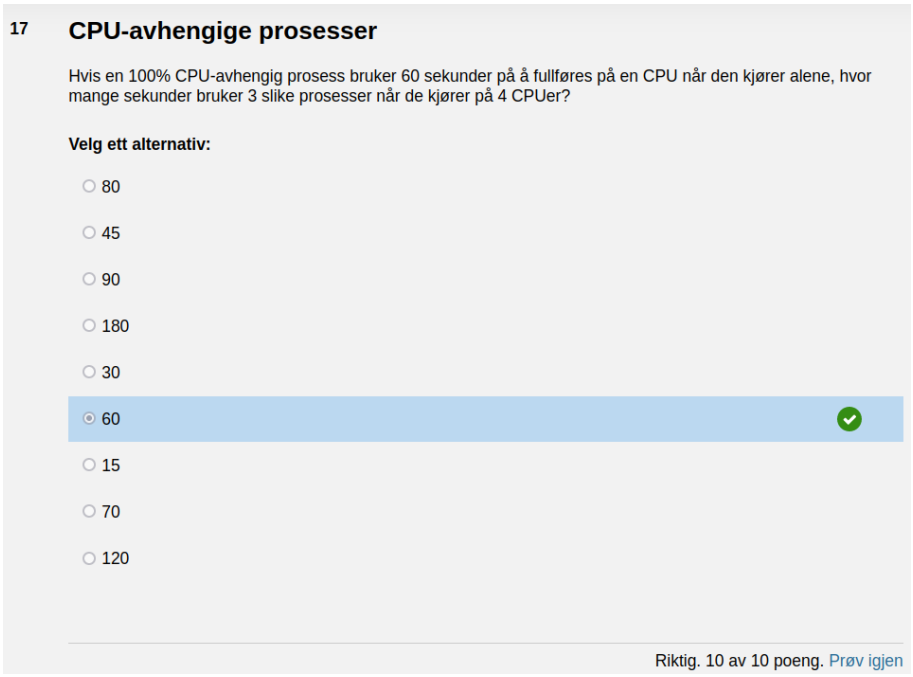
15)



16)



17)



Internminne

18)

18 **Lagringsmedier**

Hvilket av følgende utsagn er riktig om lesehastigheten fra forskjellige lagringsmedier?
Velg ett alternativ:

- Registere og RAM er omtrent like hurtig
- SSD disk er hurtigere enn cache
- L3 cache er hurtigere enn L2 cache
- Disk er hurtigere enn RAM/internminne
- Registere er hurtigere en cache ✓
- RAM er hurtigere enn cache

Riktig. 10 av 10 poeng. Prøv igjen

19)

19 **Internminne**

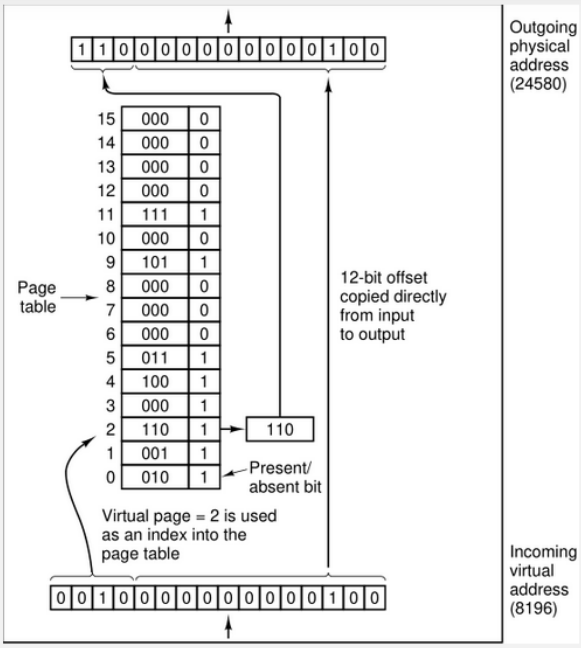
I et C-program er det et integer (int) array som har 4*1024*1024 elementer. Hvor mange byte utgjør dette arrayet?

Velg ett alternativ:

- 1 MiB
- 32 GiB
- 16 GiB
- 4 Gib
- 32 MiB
- 512 KiB
- 8 GiB
- 8 MiB
- 4 MiB
- 16 MiB ✓
- 256 KiB
- 1 GiB

Riktig. 10 av 10 poeng. Prøv igjen

20)



Velg ett alternativ:

- 24592
- 24588
- 8200
- 4108
- 8204 ✓
- 8196
- 24580
- 4096
- 12
- Gir page fault

Riktig. 10 av 10 poeng. Prøv igjen

21) Kompilering og kjøring gir følgende:

```
kan@os:~/mem$ cat mem1.c
#include <stdio.h>

int array[10000][10000];

void main(int argc, char *argv[]){
   int i,j;
   for(i = 0;i < 10000;i++){
      for(j = 0;j < 10000;j++){
 array[i][j] = 5;
      }
   }
}
```

```
kan@os:~/mem$ cat mem2.c
#include <stdio.h>

int array[10000][10000];

void main(int argc, char *argv[]){
   int i,j;
   for(i = 0;i < 10000;i++){
      for(j = 0;j < 10000;j++){
 array[j][i] = 5;
      }
   }
}

kan@os:~/mem$ gcc mem1.c
kan@os:~/mem$ time ./a.out

real 0m0.997s
user 0m0.745s
sys 0m0.252s
kan@os:~/mem$ gcc mem2.c
kan@os:~/mem$ time ./a.out

real 0m3.000s
user 0m2.683s
sys 0m0.317s
```

Kjøringen av mem2 tar tre ganger så lang tid til tross for at de begge utfører like mange instruksjoner. Dette kommer av at for mem2 blir det mange flere cache-misses fordi denne måten å løpe igjennom arrayet på også hopper over store distanser i RAM. I internminnet er slike C-arrayer organisert slik at

```
array[0][0] ligger rett før array[0][1] og array[0][2]
array[0][0] ligger langt unna array[1][0] og array[2][0]
```

For å unngå for mange cache-misses bør man derfor løpe igjennom den andre indeksen av arrayet i den innerste løkken slik det gjøres i mem1.c. I mem2.c løper man igjennom den første indeksen i den indre løkken og det vil føre til mange cahche-misses siden lokasjonene i RAM det skrives til ikke ligger etterhverandre.

Bash-scripting

22)

```
#! /bin/bash

if [ -d dir ]
then
   rm -r dir
fi

mkdir dir
cd dir
for d1 in a b c d
do
   mkdir $d1
   cd $d1
   for d2 in 1 2 3 4
```

```
   do
     mkdir $d2
     echo "dir/$d1/$d2" > $d2/fil.txt
   done
   cd ..
done

# Alternativt for løkkene:
# for d1 in {a..d}
# for d2 in {1..4}
```

PowerShell

23)

```
$f1 = $args[0]
$f2 = $args[1]
$a = (Get-ChildItem $f1).LastWriteTime
$b = (Get-ChildItem $f2).LastWriteTime
$diff = $b - $a
$days = $diff.TotalDays

"$f1 er $days dager eldre enn $f2"
```