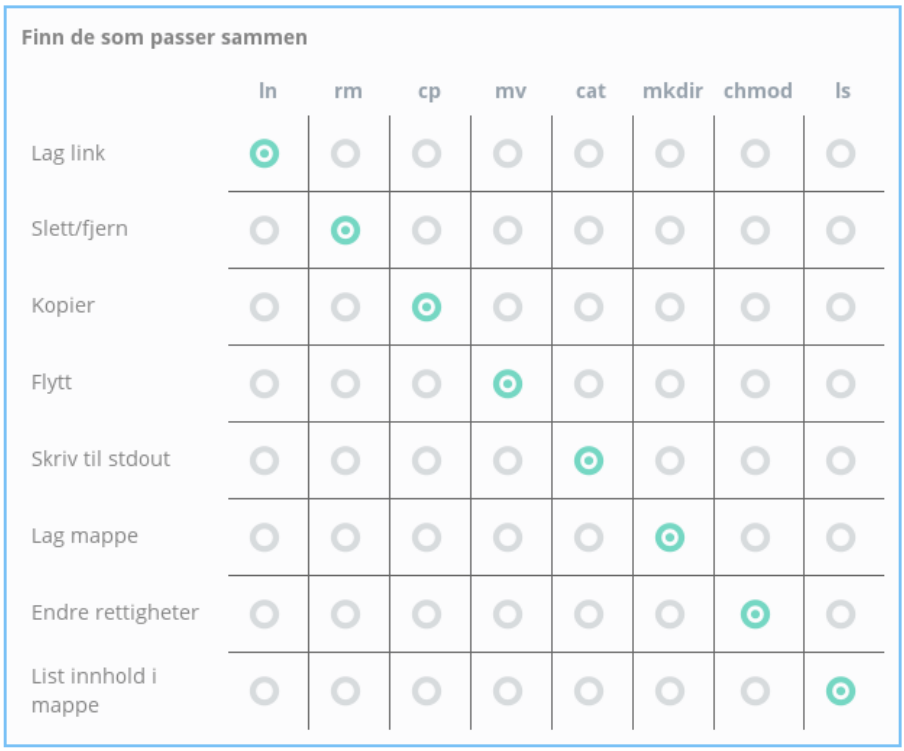
Løsningsforslag Eksamen høst 2017 Operativsystemer

Oppgave 1 - Linux kommandolinje (%)

a) `pwd`

b) `ps`

c)



Oppgave 2 - Bash-scripting (%)

a) `ping -i 5 www.hin.no`

b) `ping -c 1 www.hin.no`

c) `ping -c 1 -t 1 www.hin.no`

d) `ping -c 1 -t 1 www.hin.no | grep "Time to live exceeded"`

e) `ping -c 1 -t 2 www.hin.no | grep "Time to live exceeded"`

f)

```
#! /bin/bash

ttl=1
host=$1
while [ "true" ]
do
   res=$(ping -c 1 -t $ttl $host | grep "Time to live exceeded")
   if [ ! "$res" ]
   then
      echo "Det er $(($ttl - 1)) rutere på vei til $host"
      exit
   else
      echo "$ttl $res"
      ((ttl++))
   fi
```

Oppgave 3 - C og Assembly (%)

a)

```
gcc add.c
./a.out
```

b) `Resultat: 42`

c)

1. Tallet 13 legges i RAM som 4 bytes. (på adressen %rbp peker på, minus 8(bytes))
2. Tallet 29 legges i RAM som 4 bytes. (på adressen %rbp peker på, minus 4)
3. Tallet 29 som ble lagt i ram kopieres til registeret %eax
4. Tallet 29 som ligger i %eax legges til verdien 13 som ligger i RAM på adressen -8(%rbp)

Resultatet er tallet 42 som ligger lagret i RAM på adressen `-8(%rbp)`

d) C-instruksjonen leder til de to assembly-instruksjonene

```
movl -4(%rbp), %eax
addl %eax, -8(%rbp)
```

Siden det ikke finnes noen X86-instruksjon som legger sammen to tall i RAM, må først ett av tallene legges i et register. Generelt kan en C-instruksjon som her lede til flere Assembly-instruksjoner. Generelt tilsvarer en Assembly-instruksjon en enkelt maskin-instruksjon (men det finnes noen assembly-direktiver som ikke leder direkte til en instruksjon).

e) Opsjonen `-O` optimaliserer koden slik at den blir raskest mulig. Kompilatoren avdekker da at uansett hvordan denne koden kjøres, vil resulatet bli 42, ingen input eller andre forhold kan endre på det. Dermed lager kompilatoren ganske enkelt kode som legger 42 i et register(før tallet skrives ut). Dette er den raskest mulige koden for å gi dette resulatet.

Oppgave 4 - Serialisering og Mutex (%)

a) Kode som må fullføres uten at andre prosesser bruker samme felles ressurs

b) At OS skifter prosess som bruker CPU'en

c) Serialisering er nødvendig for at to prosesser ikke skal oppdatere felles data samtidig, noe som kan resultere i galt og utilsiktet sluttresultat.

d) I maskinkoden vil ikke disse instruksjonene bli utført med bare en instruksjon, slik som man så i "C og Assembly"-oppgaven. Anta at operativsystemet switcher fra T0 til T1 etter at T0 har lastet verdiene av `felles` og `tall` inn i hvert sitt register og T1 gjør sin oppdatering i timeslicen den har fått. Når T0 så får lov til å fortsette vil den legge sammen de to tallene den opprinnelig hadde lastet inn i registerne og legge resultatet ut i `felles`. Dermed vil T1 sin oppdatering være forsvunnet. (Hvis maskinkoden blir laget nøyaktig slik som i forrige oppgave, vil ikke problemet oppstå, da variabelen `felles` oppdateres i en enkelt instruksjon. Men problemet er at det kan variere hvordan kompilatoren velger å lage maskinkoden og den kan velge å først legge begge verdienen inn i registere, eller legge `felles` til verdien av et lokalt register og etterpå lagre resultatet, hvis det er mest effektivt. Hvis dette er tilfelle, vil problemet over kunne oppstå.)

e) Problemet unngås ved at prosessene utelukker hverandre ved å bruke mutex-funksjonene, da serialiseres utføringen av trådene og man er sikker på at en race condition ikke oppstår. Koden for henholdsvis T0 og T1 blir:

```
Get_Mutex(lock);
felles = felles + tall;
Release_Mutex(lock);
```

```
Get_Mutex(lock);
felles = felles - tall;
Release_Mutex(lock);
```

f) Koden for trådene er som i forrige deloppgave, bortsett fra at T0, som har pid = 0, kaller funksjonene med argumentet 0 og T1 kaller dem med argumentet 1. Hvis T0 kaller `Get_Mutex(0)` og turn = 1, betyr det at det er T1 sin tur til å ha tilgang til `felles` og T0 må vente til T1 er ferdig og har utført instruksjonen `turn = 1 - 1` slik at det blir T0 sin tur. Uansett når prosessene blir avbrutt av en context switch, vil kun en av dem ha tilgang til `felles`, fordi turn settes **etter** at den kritiske fasen er avsluttet.

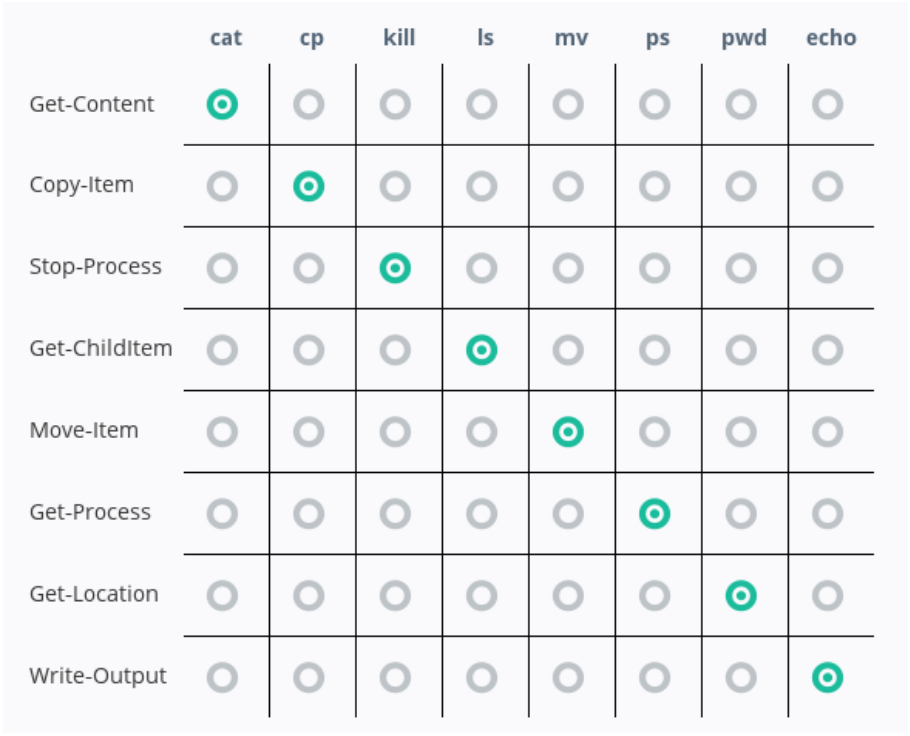
Mutex-prosedyrene inneholder en venteløkke som kaster bort CPU tid på ikke gjøre noe. Men venteløkke har f. eks. Peterson algoritmen også; den største ulempen med implementasjonen er at hvis en av trådene gjør seg fort ferdig med det kritiske avsnittet, **må den alltid** vente til den andre prosessen har vært inne i kritisk avsnitt, før den kan aksessere `felles` igjen. Og det gjør denne implementasjonen ubrukelig i de fleste tilfeller.

Oppgave 5 - Powershell (%)

a) I PowerShell sendes hele objekter med metoder og egenskaper.

b) `(ls dok.pdf).Name`

c)



d)

```
$dir = "C:\pdf"
if( ! (test-path $dir)){
   mkdir $dir
}

foreach ($fil in ls -r C:\){
   if($fil.extension -eq ".pdf"){
      $m = $fil.CreationTime.Month
      $y = $fil.CreationTime.Year
      $d = "$dir\Y$y\M$m"
      if( ! (test-path $d)){
         mkdir $d
      }
      cp $fil.FullName $d
   }
}
```

Oppgave 6 - Internminne (%)

a) At det tar like lang tid å hente en byte fra hvor som helst i minnet.

b) Ja, det vil da gå mye hurtigere å hente et byte som ligger i cache en om det ligger i RAM fordi cache-minnet er raskere og nærmere CPU.

c) Den vil bestå av 5000x5000x4 byte = 100 Megabyte. Mega betyr da en million i SI-betydningen (= 95.37 MebiByte).

d) Hele arrayet vil kunne være i minnet på en gang når programmet kjøres, så paging er ikke en faktor her. Uten cache ville da programmet gå like fort i begge tilfeller, siden det ville ta like lang tid å hente hvert matriseelement. Men i det første tilfellet ser man at arrayet skrives til i samme rekkefølge som det ligger

lagret i RAM. Dermed vil det skrives fortløpende til bytes etterhverandre i cache som så legges samlet ut i minnet. Dette går mye raskeren enn i program to hvor det er 20000 byte mellom hvert matrise element som skrives til. Dermed vil det stort sett bare bli cache-miss og det går totalt sett mye lengre tid å skrive til minnet.