

## NOTATER UKE 17 –

### *Mutex, Semaforer, Deadlock*

Kritisk avsnitt er det avsnittet i koden som er helt avgjørende når det gjelder synkronisering, og det er da typisk når man aksesserer en felles variabel.

Sist uke så vi på et eksempel med en felles variabel ‘saldo’. I den ene prosessen økte variablene med ‘mill’ og i en annen minket den med en ‘mill’. Konklusjonen er at en prosess som allerede er i et kritisk avsnitt bør få fullføre før en annen prosess kan endre også.

To prosesser P1 og P2 kjører:	
P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

Vi begynte å se på en lock-instruksjon som hindret andre prosesser i å endre variablene med pthreads, da låste vi minnebussen, bussen ut i RAM. Tråder som kjører på forskjellige CPU-er er det vanskelig å koordinere de, fordi de kjører uavhengig, det er bedre å kontrollere de på samme CPU. Selv når vi kjører på samme CPU, har vi sett at en enkelt addisjons-kodelinje fører til minst to andre instruksjoner, og det kan da oppstå en context switch midt i mellom disse, som fører til at variablene er usynkronisert.

## METODER FOR Å BEHANDLE KRITISK AVSNITT

- Direkte måte å sørge for dette er ved å skru av interrupts, da er du sikker på at det i hvertfall ikke kommer noe context switch, i hvertfall i denne CPU-en. Vi kan sikre kritisk avsnitt som bildet under:

```
disableInterrupts();  
saldo = saldo - mill;  
enableInterrupts();
```

B) Bruke en form for lås som gjør at bare en prosess av gangen har tilgang til felles data.

Dette er lik som den lock-instruksjonen som vi har sett på. Generelt kalles en slik lås:

- MUTual EXclusion = MUTEX = gjensidig utelukkelse
- mest brukt
- mange implementasjoner

## LINUX OG WINDOWS EKSEMPEL

Linux tilfelle som viser et annet eksempel på hvordan man bruker en lås. Det likner på det med lock og MUTEX også, og det er en teknikk man ofte bruker for å si ‘nå bruker jeg denne filen, du må vente til jeg er ferdig’. I Sendmail (som er et Linux mailprogram) gjøres dette ved å lage en lock til hver bruker. Hvis denne filen eksisterer, kan inbox ikke leses/skrives til. Sendmail og andre mailprogram lager denne filen før de skriver/leser mail og fjerner den når de er ferdig.

```
/var/mail/haugerud.lock
```

## WINDOWS EKSEMPEL

Under er det to tilbud til programmerere for å lage kritisk avsnitt. Disse funksjonskallene vil sørge for at etter EnterCriticalSection vil ingen andre prosesser eller CPU endre felles variablene. Det som typisk er felles ligger i RAM så igjen så vil disse funksjonskallene låse databussen. Dette forsinker alle prosesser som forsinker alle andre prosesser eller oppgaver i parallel.

Win 32 API'et har to funksjonskall

- EnterCriticalSection
- LeaveCriticalSection

## SOFTWARELØSNING AV MUTEX, GetMutex(lock)

Vi har allerede sett på lock på x86-instruksjonen lock, som vi skal se senere, for å lage en effektiv løsning så trenger man hjelp fra hardware. Men det går også an å lage softwareløsninger som selv om de har context switch, sørger for at en MUTEX.

- Trenger to funksjoner GetMutex(lock) og ReleaseMutex(lock)
- Gjør at en prosess av gangen kan sette en lock.
- Gir følgende løsning:

```
GetMutex(lock);      // henter nøkkel
KritiskAvsnitt();   // saldo -= mill;
ReleaseMutex(lock); // gir fra seg nøkkelen
```

GetMutex-metoden vil vi skal hente en lock, hente en nøkkel, og det betyr at siden denne nøkkelen er her, kan ingen andre ta den. Da kan det kritiske avsnittet fullføres, og det kan senere gis fra seg nøkkelen med ReleaseMutex.

Under er det første forsøket på å lage en Software-mutex. Vi har en statisk variabel som betyr at den kan aksesseres av to eller flere prosesser. Så har vi Getmutex(lock) med en while(lock){}. De to parentesene gjør ingenting, det er faktisk en maskininstruksjon som gjør ingenting, den heter NOP = no operation. Det den gjør, er å teste igjen og igjen om lock er true. Hvis lock er true så går den i løkka og gjør ingenting. Så igjen, det den gjør er å hele tiden sjekker om at lock er true gang på gang, og så lenge den er true, så står den og venter. Der vil den stå evig, med mindre det er andre prosesser som aksesserer denne locken og setter den til false. Hvis ingen andre har satt den til false før, så vil den første prosessen som gjør en GetMutex, den vil gå inn i koden, lock er false på while, så vil den gå inn og sette lock==true. Da er det trygt, da kan den gå inn i sitt kritiske avsnitt for den har nå gått inn, satt lock = true, akkurat som om man skal skru på en lås. Hvis en annen prosess kommer inn og også forsøker å gjøre Getmutex, før det kritiske avsnittet, så vil den komme inn og se, ‘oi det kritiske avsnittet er på, da må jeg stå og vente’, for da venter det helt til den første prosessen skal komme inn, har gjort ReleaseMutex. Når da prosess 2 er ferdig med sitt kritiske avsnitt, så vil den release mutexen og det vil den ved å sette lock = false.

```
static boolean lock = false; // felles variabel

GetMutex(lock)
{
    while(lock){}
    lock = true;
}
ReleaseMutex(lock)
{
    lock = false;
}
```

Problemet her: vi kan få et problem her hvis det kommer en context switch, hvis vi tenker oss at disse begge kjører på samme CPU, rett etter at det sjekkes om lock = false. En test i while består av flere deler. Den første delen tar for seg å hente verdien og legge det i et register. Etter å ha hentet den inn må det skje en compare, sammenlikne verdien med 0 for eksempel, etter compare må den hoppe avhengig av verdien. Poenget her er: hva om det skjer en context switch rett etter at den har hentet inn verdien.

Jo, da fryses selve prosessen. Det neste den vil gjøre er å sette lock = true og gå inn i kritisk avsnitt. Men når denne fryses og lock nr 2 kommer inn, er den fortsatt false, så prosess nr 2 vil enkelt og greit bare hoppe over løkka og sette lock = true. Men da er det for sent for da er prosess 1 allerede inne i kritisk avsnitt. Så kommer prosess 2 etter og vil også gå inn i kritisk avsnitt. Da er vi tilbake til det gamle problemet, hvis tilfellet nevnt skjer, er begge og driver innenfor det kritiske avsnittet samtidig. Denne løsningen vil dermed fungere godt før dette unntaket kan skje.

## HARDWARE-STØTTET MUTEX

I de løsningene vi har sett på, krever litt kode, i tillegg til at alle krever busy-waiting. Og det handler om det med at man står og venter, og det bruker CPU med NOP-instruksjonen, det gjør at man står og venter. I praksis brukes oftest hardwarestøttede løsninger. En slik støtte kan lages med egen instruksjon om man har **testAndSet** (TSL). Det er en instruksjon som gjør begge operasjonene som vi snakket om, nemlig om å teste den og endre den i en og samme instruksjon. Det er viktig med en og samme instruksjon slik at det ikke kan komme noe context switch mens den instruksjonen utføres.

En testAndSet vil uansett låse minne-bussen slik at ikke andre CPUer kan endre eller lese verdien. Da kan man implementere getMutex på følgende måte:

```
GetMutex(lock)
{
    while(testAndSet(lock)) {}
}
```

En context switch kan ikke ødelegge siden testen og endringen av lock skjer i samme instruksjon.

Siden denne instruksjonen også låser minnebussen er man bombesikker på at ingen andre interrupt kan finne sted.

## X86-LOCK

Denne utføres før en kritisk situasjon. Sist så vi dette med først lock også en add. Her også vil man låse minnebussen slik at ingen andre kan få tilgang til den verdien, den minneadressen som man bruker. Alle andre minneadresser kan brukes, men ikke den. Det sikres at instruksjonen etter lock som utføres på en variabel i minne får avsluttet hele sin operasjon uten at RAM endres. Det kritiske avsnittet fullføres før noen andre tråder slipper til. Fungerer kun for et kritisk avsnitt som består av en enkelt instruksjon.

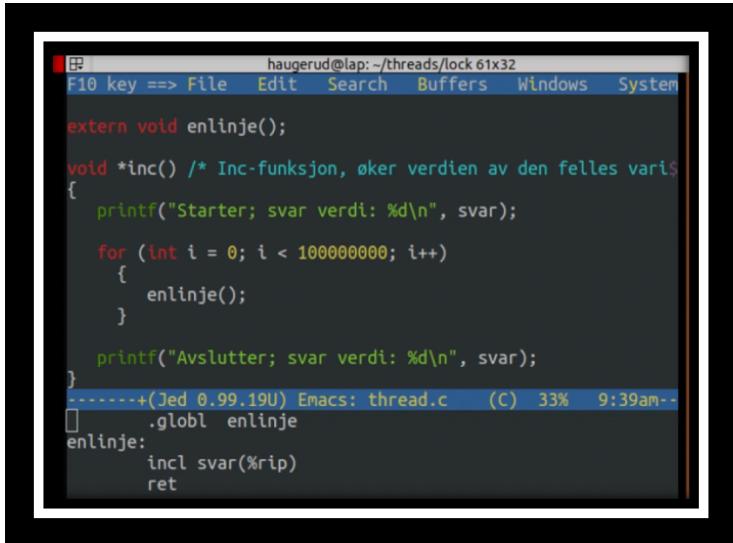
Spørsmål: Virker testAndSet også for flere CPU-er? -Ja

## DEMO: LOCK OG PTHREADS

Repetisjon fra forrige uke. Nå skal vi se på hva som skjer om vi skriver endringen i to operasjoner. Med gcc sin kompilerte kode i flere deler for å øke variablene skal vi se at man også kan få et problem hvis det oppstår en context switch. At man tvinger begge til å kjøre på samme CPU, men det kan skje feil grunnet de flere operasjons-linjene.

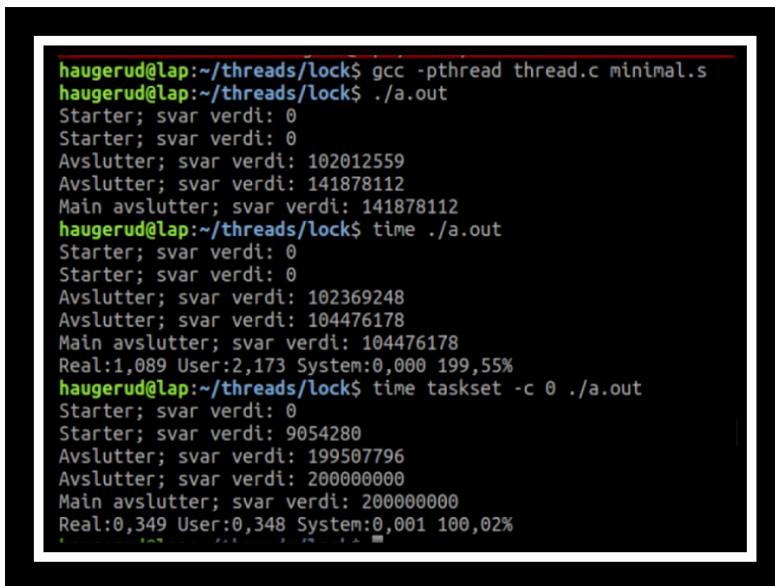
## DEMO: SAMME CPU, 3 LINJER FOR Å ØKE SVAR

Kompilere disse to programmene som kjører 2 tråder som hele tiden utfører enlinje(). Enlinje() er bare å øke svar med en, så når begge trådene gjør dette hundre millioner ganger skal svaret bli to hundre millioner ganger. Svaret ble aldri det samme grunnet usynkronisering.



The screenshot shows an Emacs window with a dark theme. The buffer contains assembly code for threads. The code includes declarations for `enlinje()` and `inc()`, and a main loop that calls `enlinje()` 100 million times. The assembly code at the bottom shows instructions like `.globl enlinje`, `enlinje:`, `incl svar(%rip)`, and `ret`.

Under ser vi at da det kompileres og kjøres er det ulike svar fordi threadsa ikke er synkroniserte, men svaret vil bli riktig hvis de kjøres på samme CPU.



The terminal window shows the compilation of `thread.c` to `minimal.s` and `a.out`. It then runs `a.out` twice. The first run shows variable values starting at 0 and increasing to approximately 141878112. The second run shows values starting at 0 and increasing to approximately 104476178. Finally, it runs `taskset -c 0 ./a.out`, which completes much faster, with variable values starting at 0 and increasing to approximately 200000000. Timing information is also provided for each run.

Det som kan være en grunn til at det går raskere ved å kjøre på samme CPU er caching (skal sees på neste uke). Caching vil si å mellomlagre verdien.

En linje assemblykode → en linje maskinkode.

En linje høynivåkode → flere linjer maskinkode.

Under kompileres kode med minimal2 som inneholder en del koder hvor man henter inn kode eksplisitt fra Ram, også økes den også legges den igjen. Hvis det kommer en context switch før den har økt verdien, kommer den andre tråden som henter inn svaret på nytt.

```

haugerud@lap:~/threads/lock$ gcc -fno-stack-protector -o thread thread.c
haugerud@lap:~/threads/lock$ ./thread
Starter; svar verdi: 0
Starter; svar verdi: 8895034
Avslutter; svar verdi: 167805555
Avslutter; svar verdi: 169867292
Main avslutter; svar verdi: 169867292
haugerud@lap:~/threads/lock$ ./thread
Starter; svar verdi: 0
Starter; svar verdi: 8834249
Avslutter; svar verdi: 165803877
Avslutter; svar verdi: 168818869
Main avslutter; svar verdi: 168818869
haugerud@lap:~/threads/lock$ ./thread
I

```

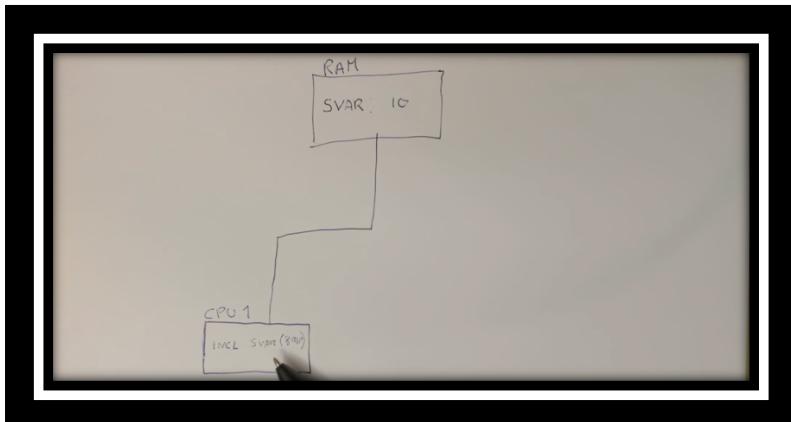
```

extern void enlinje();
void *inc() /* Inc-funksjon, øker verdien av den felles variabelen */
{
    printf("Starter; svar verdi: %d\n", svar);
    for (int i = 0; i < 100000000; i++)
    {
        enlinje();
    }
    printf("Avslutter; svar verdi: %d\n", svar);
}
-----+(Jed 0.99.19U) Emacs: thread.c  (C) 33% 9:47am--.globl enlinje
enlinje:
    movl svar(%rip), %eax
    incl %eax
    movl %eax, svar(%rip)
    ret

```

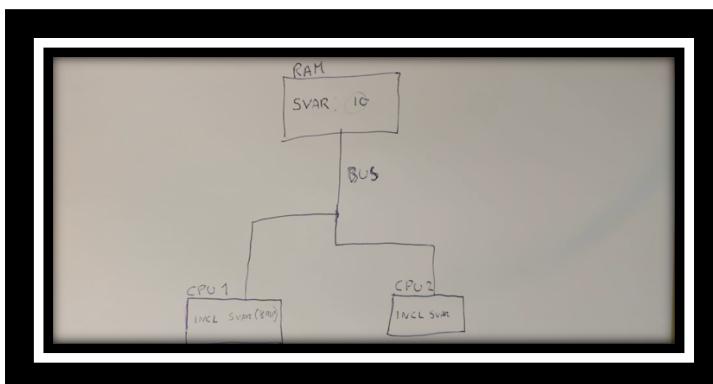
## DEMO 2 CPU-ER OG LÅSING AV MINNEBUSS

Vi ser en CPU1 og en buss som går til RAM oppe. Opp i RAM har vi en variabel som heter svar, som nå er 10.

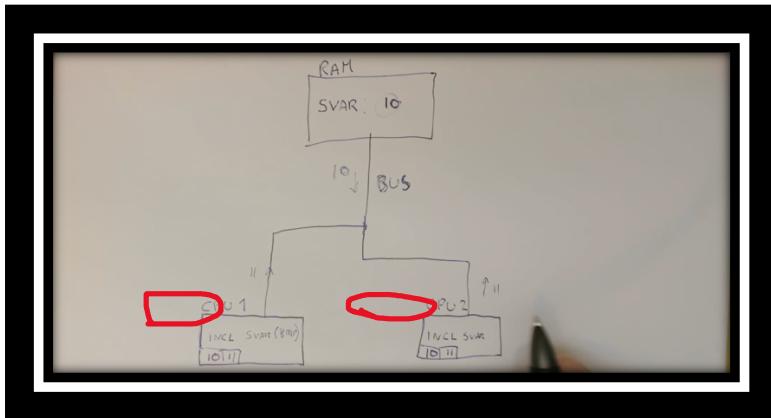


I CPU1 da har vi en instruksjon som sier incl svar(%rip), altså incline long til svar.

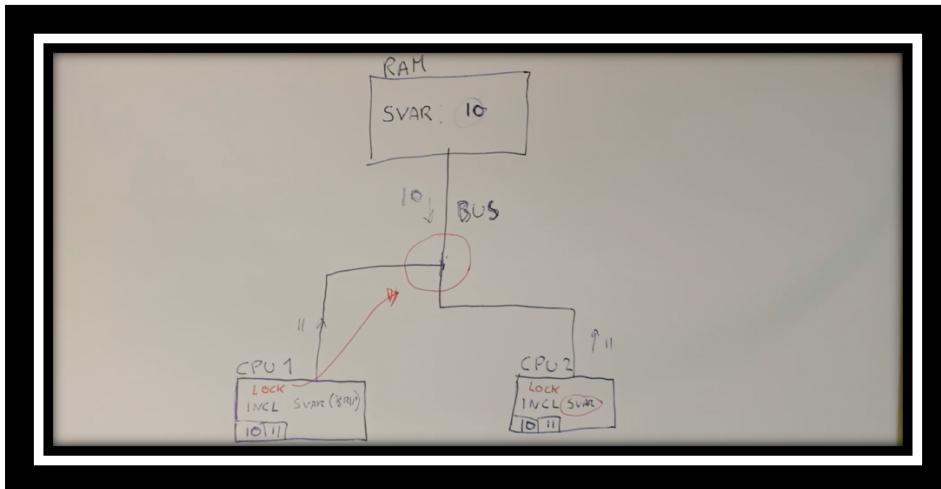
Instruksjonen må på en eller annen måte først hente verdien også øke. Også legger vi til enda en CPU, CPU2, som også er koblet med bussen opp til RAM.



Da er problemet hvis de CPU-ene ikke er koordinerte, så opererer de samtidig med å hente variablene svar fra RAM også sender de med instruksjonen inn i ALU-en for å regne også sende variablene tilbake til RAM. Da får vi allerede et problem om de gjør det her samtidig.



Over er det mulig å se at variablen hentes til begge CPU-ene for å så sendes til ALU hvor det legges til 1, også lagres det (rød sirkel). Også skal det sendes tilbake. Uansett hvilken som kommer først tilbake vil 11 lagres, da det skal være 12.

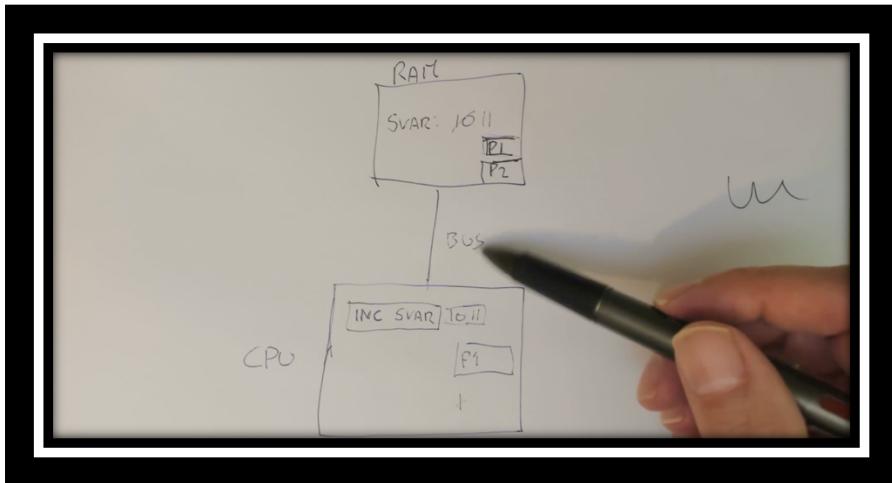


Som Hårek da over skriver på er lock variabler foran CPU-ene, slik at databussen låses og at ingenting kan komme i veien når den driver med sitt. CPU1 får utføre sin instruksjon i fred og ro med å hente inn 10, øke verdien med 1 med hjelp fra ALU, så sender den variablene tilbake, slipper opp låsen. Så skal CPU2 fortsette å sjekke om det fortsatt er låst, og når den ser at det ikke er det så vil den hente ut verdien fra RAM som nå er 11, få ALU til å øke verdien med 1 til og da legges verdien 12 tilbake i register-adressen i RAM.

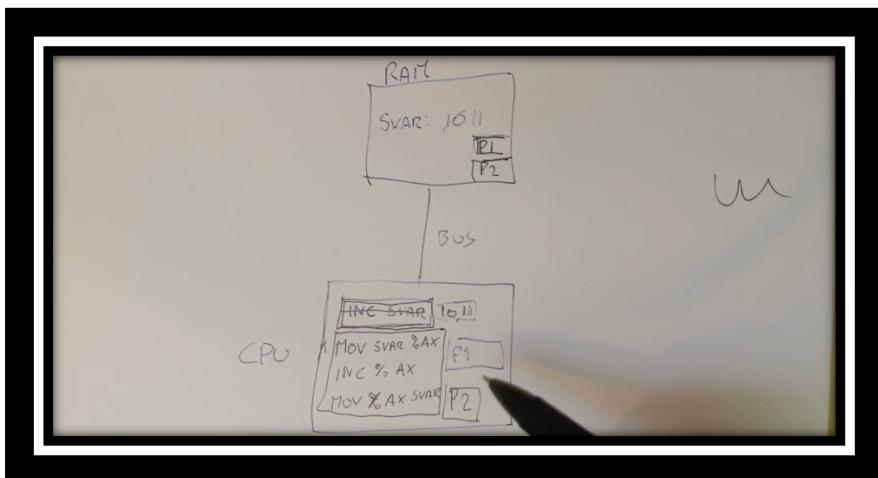
### CPU OG TASKSET SLIK AT BEGGE TRÅDER KJØRER PÅ SAMME CPU

Da har vi bare en CPU, og RAM oppe med svar variablen lik 10. la oss si CPU-en har en instruksjon med INC svar, da hentes tallet 10 med databussen, ALU får 10, øker med 1 og

sender tilbake. Men la oss si at vi har to prosesser, eller to tråder som kjører på samme CPU. Da vil det være p1 og p2, og de har et lite område i RAM, hvor de har sine verdier lagret. Og det er når man gjør en context switch at man henter inn de verdiene



Når vi istad kjørte taskset, og hadde en instruksjon så vi at det ikke hadde noe å si at en har flere tråder tilstede. Hva om vi nå bruker koden med det andre tilfellet der flere maskinkodelinjer er tilstede. Da vil det være trøbbel på tross av taskset. Da har vi to prosesser P1 og P2, og dette likner litt med tilfellet hvor 'mill' skulle økes og minkes.



Ja, selvom det bare er en CPU og en buss oppstår det konflikter. La oss si vi har 11 i svarvariabelen i RAM, også er det P1 som kjører. P1 henter inn verdien og legger den i %AX. Også kommer det en context switch, da vil verdien 11 lagres i PCB-en for P1. så starter P2 å kjøre som da også skal utføre den første instruksjonen ved å hente inn variablen fra RAM, øker den i ALU og legger den ut tilbake igjen, og den vil da kunne gjøre det flere ganger før det kommer enda et context switch.

## DEMO: MUTEX OG IMPLEMENTASJON AV SEMAFOR I OS

Semaforer er et begrep som er brukt mye når det gjelder synkronisering. Det er en slags teller, så en binær semafor, altså en semafor som bare har 0-er eller 1-er, er akkurat som en Mutex. Generelt kan en semafor ha flere verdier. En kan for eksempel starte med  $S = 10$ , da har man 10 resurser, som man kan bruke opp før man må vente og synkronisere med andre.

En semafor er en integer  $S$  som signaliserer om en ressurs er tilgjengelig. To operasjoner kan gjøres på en semafor:

```
Signal(S): S = S + 1; # Kallas ofte Up(),  
Wait(S): while(S <= 0) {}; S = S - 1; # Kallas ofte Down()
```

Signal og wait må være uninterruptible og implementeres med hardwarestøtte eller i kjernen for å være atomiske (umulige å avbryte).

Signal (S): signaliserer at alt er klart. Signal og wait må være uninterruptible og implementeres med hardwarestøtte eller i kjernen for å være automatiske (umulig å avbryte).

Når man tar Wait(S) brukes dette på samme måte som en lock og før Signal(S) kan man ha et kritisk avsnitt inni.

Binær semafor  $S = 0$  eller  $1$  (som lock) (initialisert til  $1$ )  
Teller semafor  $S$  vilkårlig heltall (initialisert til antall ressurser)

En semafor initialisert til  $S=1$  kalles ofte en mutex: Kan da brukes slik til å takle et kritisk avsnitt:

```
Wait(S);  
KritiskAvsnitt();  
Signal(S);
```

Operativsystemet har den store fordelen om at det kan styre prosesser og sette dem inn og ut av køer. Under vises en implementasjon av semafor i OS:

Hvis en semafor implementeres i OS kan busy waiting unngås.

```
Signal(S){  
    S = S + 1;  
    if(S <= 0){  
        wakeup(prosess);  
        # Sett igang neste prosess fra venteliste  
    }  
}  
  
Wait(S){  
    S = S - 1;  
    if(S < 0){  
        block(prosess);  
        # Legg prosess i venteliste  
    }  
}
```

5:39

Hvis vi starter med Wait: hvis en ressurs er opptatt og man må vente kan OS blokkere dette og sette den i en venteliste. Det fine med det er at da tas bare prosessen ut av ready list, da trenger den ikke å stå ute i lufta å kjøre med en busy-waiting. Når andre prosesser er ferdig og gjør et signal, da kan vi vekke opp andre prosesser fra ventelisten (se Signal(S) over).

OS legger til rette at programmerere kan synkronisere med signal og wait, men da må man skrive kode som gjør dette, OS legger til rette ved vår bruk av semaforer.

## DEADLOCK

Vi har først sett på multitasking og parallelisering, altså å få ting til å gå fort ved å få ting til å kjøre parallellt. Men så kom vi til noen tilfeller hvor vi innså at vi må serialisere koden, det blir rett og slett ødeleggende når kode kjører i parallel, spesielt dersom et eksisterer felles variabler, kalles kritisk avsnitt. Et eksempel vil være ved to eller flere tråder som kjører med hverandre, dermed kan vi konkludere med at et system må være thread-safe. Dermed må vi serialisere og synkronisere, og vi har nettopp sett ulike måter å serialisere på: mutex, semaforer og monitorer. Dette er tilbud fra OS til en programmerer for å serialisere. Monitorer som vi så i java, er enklere å forholde seg til. Med en enkelt metode, ved å legge rundt en blokk, som den Synchronize-metoden, og den vil da synkronisere hele metoden på en veldig enkel metode.

Når man serialiserer på den måten, med mutexer og semaforer, tvinger prosessorer til å vente på hverandre, så får man et problem med at **deadlocker kan oppstå**. Det må være tre klare kriterier tilfredsstilt for at deadlock kan oppstå. Hvis man klarer å unngå alle disse kriteriene,

kan man unngå deadlock, og oftest koder man for at deadlock ikke skal oppstå fordi man ikke vil ha det i et program.

1. det første er at man må ha en form for mutex, altså man har ressurser som ikke kan deles.
2. også må man også ha et kriterie på å (at en prosess kan) beholde sin ressurs mens den venter på andre.
3. en prosess kan ikke tvinges til å gi opp sin ressurs (felles minne, disk, etc). Med 1, 2 og 3 oppfylt, kan deadlock oppstå ved sirkulær venting.

Andre webressurser sier at kriterier for at en deadlock kan oppstå sees som:

1. mutual exclusion: minst en ressurs må være under eksklusiv kontroll. Dette betyr at bare en prosess kan ha tilgang til ressursen om gangen.
2. hold and wait: en prosess som allerede holder en ressurs, kan kreve tilgang til en annen ressurs og samtidig holde ressursen som allerede er tildelt til den.
3. no preemption: ingen ressurser kan tas fra en prosess uten dens samtykke, og prosessen vil holde på ressursene til den er ferdig med å bruke dem.
4. circular wait: det må være en syklus av venting der hver prosess venter på en ressurs som er tildelt til den neste prosessen i syklusen.

Under er et enkelt eksempel på deadlock. Generelt kan man få deadlock om to eller flere prosesser venter på hverandre. I det eksemplet her ser vi at vi kjører en prosess A og en prosess B. Wait er for å vente på en ressurs og signal er for å avfinne den ressursen. Men da kan vi få et deadlock-problem, fordi både PA og PB prøver å få tak i en ressurs. Først gir prosess A en wait for S1 mens prosess B gjør wait for S2. mens den venter på S1 venter den også på S2. og samtidig gjør prosess 1 det samme, wait på S1. da får vi en låst situasjon på eksempel 2 i bildet under. Prosess A venter på at prosess B skal release S2, og tilsvarende venter PB på at PA skal release S1. her står det med busy-waiting, det står og spinner og spinner. Dette kalles en deadlock fordi den aldri vil løse seg opp.

To eller fler prosesser venter på hverandre, ingen kommer videre.  
 Eks. 1: P1 venter på P2, P2 venter på P3 og P3 venter på P1  
 (sirkulær venting)

Eks. 2: Deadlock med to semaforer S1 og S2, initialisert til 1:

PA	PB-kode
Wait(S1)	
Wait(S2)	Wait(S2)
.	.
Signal(S1)	Signal(S2)
Signal(S2)	Signal(S1)

## DINING PHILOSOPHERS PROBLEM

Er et kjent problem hvor deadlock kan oppstå, er kjent fordi det brukes som et eksempel problem. Man bruker mutex og semaforer osv også skal man kode dette på en måte slik at deadlock unngås. Tegningen skal fremstille et bord med fem fat spaghetti og mellom hvert fat befinner det seg en gaffel. Noen ganger sitter de og tenker, altså det spinner i luften, ingen gafler. Hvis det sitter en ved bordet er det to gafler han kan ta, systemet er slik at for å kunne spise må en ha to gafler. Med mutex osv må det dermed da kunne sjekkes om gaflene er ledige. Først en gaffel, så en til også når de kommer igang kan de spise. Problemstillingen er da å kunne programmere dette på en måte slik at alle kan spise. Igjen må programmet være slik at gaffelene bør plukkes, og det bør spises med flyt. Det kan hende at dette går veldig fint gode stunder også plutselig oppstår det deadlock. Hvis man ikke programmerer det riktig kan det hende at vi ender opp med at en person sitter med gaffelen i en hånd og alle andre bare sitter og venter for evig.

- Filosoftilstander:
- ① tenker (uten gafler)
  - ② Spiser spaghetti med 2 gafler

Tar opp en gaffel av gangen.

**Problem** Programmer en filosofprosess slik at 5 prosesser kan spise og tenke i tidsrom av varierende lengde og dele ressursene (gaflene) **uten** at deadlock (vranglås) kan oppstå.



## LØSNINGER FOR DEADLOCK

Det først man kan gjøre er å forhindre det. Det må forhindres internt i OS-kjernen. Det er umulig å forhindre bruker-deadlock. Det andre man kan gjøre er å løse opp deadlock, og dette er generelt vanskelig å kode. Det siste som er det vanligste innenfor de fleste OS-ene er å ignorere problemet.