Kritisk avsnitt

To prosesser P1 og P2 kjører:

P1-kode	P2-kode
static int saldo;	static int saldo;
.	.
.	.
.	.
saldo = saldo - mill;	saldo = saldo + mill;

- Utregningen av saldo er et **kritisk avsnitt** i koden til P1 og P2.
- Kritisk avsnitt **må fullføres** av prosessen som utfører det uten at andre prosesser slipper til.
- Det medfører at prosessene må serialiseres.

Metode A

- Skru av interupts før kritisk avsnitt
- Skru på interupts etter kritisk avsnitt

```
disableInterupts();
saldo = saldo - mill;
enableInterupts();
```

OK for en OS-kjerne, men for farlig for brukerprosesser; de kan ta over styringen.

Metode B

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

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

Linux-eksempel

File-lock for Linux-mail: hvis filen

`/var/mail/haugerud.lock`

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 ferdige.

Windows-eksempel

Win 32 API'et har to funksjonskall

- EnterCriticalSection
- LeaveCriticalSection

som applikasjoner kan kalle før og etter et kritisk avsnitt.

Softwareløsning av 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 nøkkel
```

Software-mutex, forsøk 1

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

GetMutex(lock)
   {
   while(lock){} // venter til lock blir false
   lock = true;
   }
ReleaseMutex(lock)
   {
   lock = false;
   }
```

Dette burde sikre at to prosesser ikke er i kritisk avsnitt samtidig?

Hardware-støttet mutex

- alle softwareløsninger innebærer mange instruksjoner i tillegg til busy-waiting, som koster CPU-tid.
- I praksis brukes oftest hardwarestøttede løsninger
- Kan lages med en egen instruksjon **testAndSet** (TSL).
- Tester og setter en verdi i samme maskininstruksjon.
- Låser minne-bussen slik at ikke andre CPUer kan endre eller lese verdien
- GetMutex() kan da implementeres med:

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

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

X86-instruksjonen lock

- Maskin-instruksjonen `lock` sørger for at i det korte tidsrommet neste instruksjon utføres, låses minnebussen slik at instruksjoner på andre CPUer ikke samtidig kan hente eller lagre noe i RAM
- Sikrer 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

Semaforer

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

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

Semafor og mutex

Binær semafor S = 0 eller 1 (som lock) (initialiseres til 1)

Teller semafor S vilkårlig heltall (initialiseres til antall ressurser)

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

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

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
    }
}
```

Bruk av semafor i kritisk avsnitt

Anta at semaforen S brukes til å beskytte en felles ressurs (variabel eller lignende) i et kritisk avsnitt. Prosess A og B må da kall Wait() før og Signal() etter kritisk avsnitt.

PA	PB-kode
A1	B1
A2	Wait(S)
A3	K1
Wait(S)	K2
K1	K3
K2	Signal(S)
K3	B2
Signal(S)	B3
A4	B4

Bruk av semafor i kritisk avsnitt

A	B	S
	B1	1
	Wait(S)	0
	K1	0
A1	$\Longleftarrow$ Context Switch	0
A2		0
A3		0
Wait(S)	OS legger A i kø	-1
Context Switch $\Longrightarrow$	K2	-1
	K3	-1
	Signal(S)	0 (A ut av kø)
	B2	0
	B3	0
	B4	0
K1	$\Longleftarrow$ Context Switch	0
K2		0
K3		0
Signal(S)		1
A4		1

Bruk av semafor til å synkronisere to prosesser

- Anta prosess B (PB) må vente til prosess A (PA) er ferdig med noe i sin kode
- først da kan den kan gå videre (med kodelinje B2 i eksempelet).

PA	PB-kode
A1	B1
A2	Wait(S)
A3	B2
Signal(S)	B3
A4	B4

Med en semafor S initialisert til 0, kan de da synkroniseres som følger:

B når først frem

A	B	S
A1		0
A2		0
$\Longrightarrow$	B1	0
	Wait(S)	-1
A3	$\Longleftarrow$	-1
Signal(S)		0
A4		0
$\Longrightarrow$	B2	0
	B3	0

A når først frem

A	B	S
A1		0
A2		0
A3		0
Signal(S)		1
A4		1
$\Longrightarrow$	B1	1
	Wait(S)	0
	B2	0
	B3	0

Låse-mekanismer brukt i Linux-kjernen

Linux-kjernen kan selv skru av og på interrupts for å sikre at korte kode-biter ikke blir avbrutt. Flere CPUer kan samtidig kjøre kjerne-kode, derfor er låser mye i bruk for å unngå at datastrukturer aksesseres samtidig.

Atomiske operasjoner Operasjonen kan ikke interruptes, eksempel: `atomic_inc_and_test()`

Spinlocks Mest brukt. For korte avsnitt. Bruker busy waiting.

Semaforer Kjernen sover til semaforen blir ledig igjen om den er opptatt.

Reader/Writer locks Samtidige prosesser kan lese, men bare en CPU av gangen kan skrive

Monitorer og Java synkronisering

- Monitorer er en del av et programmeringsspråk
- hele metoder eller kodeblokker synkroniseres
- Kun en monitor-metode kan kjøre av gangen
- Enklere for programmereren!

I Java implementert med `synchronized(objekt)`:

```
public static int saldo; // Felles variabel, gir race condition
public static Object lock = new Object(); //
// Argumentet til synchronized må være et objekt

            synchronized(lock)
            {
                saldo += 1; // Synkronisert kode-blokk
            }
```

Java monitor

```
$ javap -private -c SaldoThreadS
      11: getstatic     #16      // Field MAX:I
      14: if_icmpge     88
      17: getstatic     #17      // Field lock:Ljava/lang/Object;
      20: dup
      21: astore_2
      22: monitorenter
      23: getstatic     #18      // Field saldo:I
      26: iconst_1
      27: iadd
      28: putstatic     #18      // Field saldo:I
      31: aload_2
      32: monitorexit
      33: goto          41
```

Java synkronisering av metode

En løsning på vårt problem kunne være å gjøre følgende:

```
private static synchronized void upSaldo()
   {
        saldo++;
   }

 private static synchronized void downSaldo()
   {
         saldo--;
   }
```

og bruke disse metodene istedet for å endre variabelen direkte. Da ville også koden bli threadsafe og alltid gi saldo lik null til slutt. Hvorfor er det viktig at denne metoden er static?

Message passing

- Konkurrerende tråder eller prosesser synkroniseres da ved å sende signaler til hverandre.
- Virker også i distribuerte systemer
- Ulempe: ikke like effektivt som semaforer og monitorer.

Kriterier for at deadlock kan oppstå

1. Mutex: ressurser som ikke kan deles
2. En prosess kan beholde sine ressurser 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!

Deadlock

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)	
	Wait(S1)
.	.
.	.
.	.
Signal(S1)	Signal(S2)
Signal(S2)	Signal(S1)

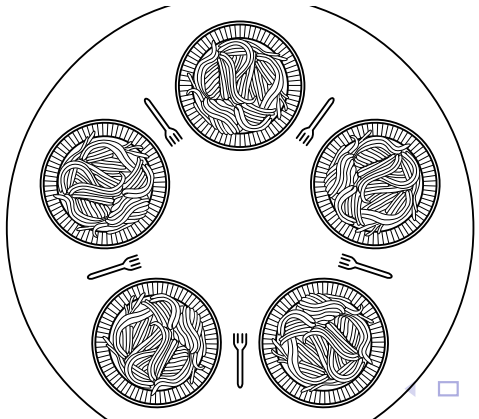
Dining Philosophers Problem

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.



Mulige løsninger for deadlock

1. Forhindre. Internt i OS-kjernen må deadlock forhindres. Umulig å forhindre bruker-deadlock.
2. Løse opp deadlock. Generelt vanskelig.
3. Ignorere problemet (mest vanlig metode for et OS).