

Obligatorisk oppgave nr 4 DAT103

Oppg 1

a) Operativsystem: Tilbyr et miljø å starte programmer og tjenester for andre program og brukere. Tjenester for brukeren kan være f.eks. Programutføring, I/O operasjoner, håndtering av filsystemet, kommunikasjon, feilfinning eller Brukergrensesnitt GUI/CLI. GUI (Grafisk brukergrensesnitt) er en brukervennlig metafor for grensesnitt og tar i bruk mus/tastatur etc mens en CLI (Command-line-interpreter) lar brukeren utføre kommandoer. Andre Systemtjenester kan være håndtering av resurser som blandt annet ressursalokering og ressursregnskap.

b) Prosess: En prosess er et program under utføring som danner grunnlaget for all beregning. Prosessutføring er sekvensiell. Et program er en passiv del som er lagret på disk mens en prosess er aktiv (altså utføringen). Programmet blir en prosess når den lastes inn i hukommelsen. Et program kan bli flere prosesser og utføres via museklikk eller kommandolinjekall.

Prossesser endrer tilstand under utføring:

ny: Prosessen opprettes

kjører: instruksjoner utføres

venter: programmet venter på en hendelse

klar: prosessen venter på en prosessor

avsluttet: prosessen er ferdig utført

En prosess kan også ha en eller flere underprosesser kalt childs. En prosess kan være uanhengig eller samarbeidende. Uavhengige prosesser kan ikke påvirke/bli påvirket av andre prosesser og det motsatte gjelder for samarbeidende prosesser. Samarbeid er kalt IPC og har fordeler i informasjonsdeling, raskere beregninger modularitet og praktisitet.

c) Forskjellen på en tråd og en prosess er egentlig mest knyttet til at opprettelsen av en tråd er lett mens opprettelsen av en prosess er tung. Dermed har tråder flere fordeler når det kommer til effektivitet og forenkling av kode. Tråder opprettes inni en applikasjon mens prosesser starter ved oppstart av et program. Å endre konteksten til en tråd er også enklere enn for en prosess. Eksempler på tråd oppgaver kan være oppdatering av GUI, svare på nettverksforespørsler, stavekontroll, henting av data osv.

d) Anta et system med n prosesser ($p_0, p_1, \dots, p_{(n-1)}$). Hver prosess har kritiske regioner der den f.eks. kan endre felles variabler, oppdatere tabeller, skrive til fil osv, men når en prosess er i en kritisk region, kan ingen andre være i sine kritiske regioner. En protokoll kan løse problemet med kritiske regioner. Hver prosess spør om lov til å gå inn i en kritisk region i en inngangsregion. Deretter utfører den og forteller andre når den er ferdig og kan så fortsette.

Det finnes flere måter å løse kritisk region problemer på og for at man skal løse problemene må følgende krav være tilfredstilt: gjensidig utelukking, altså hvis en prosess er i sin kritiske region får ingen andre prosesser være i sine. Fremdrift, altså at det blir gjort en utvelgelse slik at ingen prosesser må vente i det uendelige og sist men ikke minst Begrenset venting, altså at det finnes en grense for antall ganger andre prosesser får gå i sine kritiske regioner når en prosess har gjort en forespørsel om å gå i sin. Løsningene er også avhengig av om kjernen er avbrytbar eller ikke.

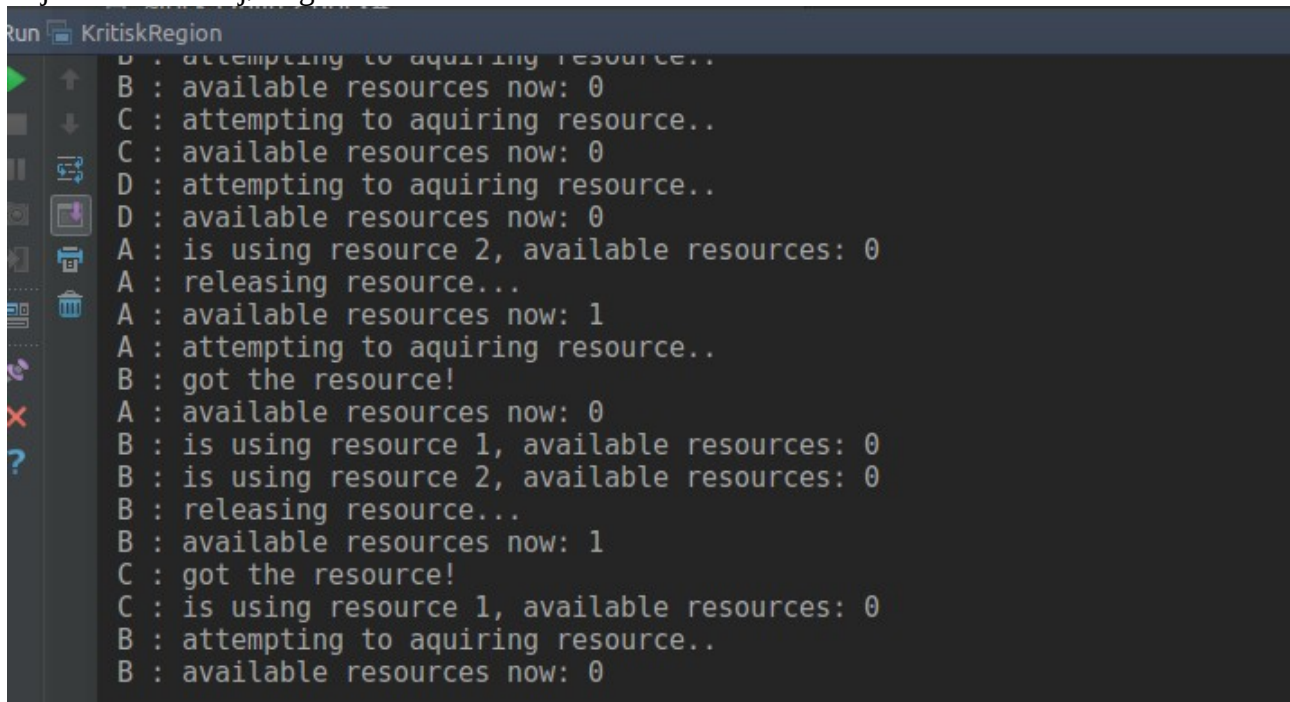
e) Semaforer er et synkroniseringsverktøy som ikke krever aktiv venting. En semafor S er en heltallsvariabel og har to operasjoner, wait() og signal(). Semaforer er lite komplisert og S kan bare påvirkes av to atomiske operasjonene som følger:

```
void wait (S) {  
    while(S <= 0)  
        ; // busy wait  
    S--;  
}  
og  
void signal (S) {  
    S++;  
}
```

En semafor kan brukes som en telle-semafor (heltall), eller en binærsemafor (0 og 1). Den kan brukes istedet for mutex lås i systemer som ikke har dette for gjensidig utelukking. en tellesemafor kan implementeres ved en binær semafor, kan løse mange synkroniseringsproblemer som blant annet bounded buffer problem, readers-writers problem og the dining-philosophers problem.

f) se filen KritiskRegion.java

Skjermskudd av kjøring:



```
Run KritiskRegion  
B : attempting to acquiring resource..  
B : available resources now: 0  
C : attempting to acquiring resource..  
C : available resources now: 0  
D : attempting to acquiring resource..  
D : available resources now: 0  
A : is using resource 2, available resources: 0  
A : releasing resource...  
A : available resources now: 1  
A : attempting to acquiring resource..  
B : got the resource!  
A : available resources now: 0  
B : is using resource 1, available resources: 0  
B : is using resource 2, available resources: 0  
B : releasing resource...  
B : available resources now: 1  
C : got the resource!  
C : is using resource 1, available resources: 0  
B : attempting to acquiring resource..  
B : available resources now: 0
```

g) Banksjefens algoritme kan beskrives som en vranglås unngåelse algoritme som er mindre effektiv enn ressurs allokeringens graf systemet men kan brukes på et system med flere ressurser med flere instanser.

Følgende krav må alltid være oppfylt i banksjefens algoritme:

- enhver prosess som kommer inn i systemet må deklarerere et maks behov instanser for hver ressurs i systemet, de kan ikke overgå maks antall instanser tilgjengelig.

- når en prosess ber om et sett med ressurser må systemet avgjøre om tildelingen av ressursene etterlater systemet i en safety state og hvis dette er tilfelle blir ressursene allokert, hvis ikke må prosessen vente til en annen prosess har frigjort ressurser.

Følgende datastruktur må vedlikeholdes i algoritmen:

Available: en vektor av lengden m der $Available[j] = k$ tilsvarer at for ressurs R_j er k instanser tilgjengelig.

Max: en $n * m$ matrise der $Max[i][j] = k$ tilsvarer at prosess P_i har et maks behov av k instanser av ressurs R_j .

Allocation: en $n * m$ matrise der $Allocation[i][j] = k$ har en prosess P_i allokert k instanser av ressurs R_j .

Need: en $n * m$ matrise der $Need[i][j] = k$ der en prosess P_i har behovet av k instanser av ressurs R_j .
Tilsvarende også $Max - Allocation = Need$.

Safety algoritmen som bankers algoritme må tilfredstille til enhver tid kan da beskrives som følger hvor man følger hvert steg helt til man ender på slutten av steg 4 i en sikker tilstand.

1. Vi lar *work* og *finish* være vektorer av lengden n og m henholdsvis og setter i første steg $work = Available$ og $finish[i] = false$ for alle i ($1, 2, \dots, n-1$). Og fortsetter så til steg 2.
2. Vi finner en index i slik at $finish[i] = false$ & $Need_i \leq Work$. Hvis dette ikke finnes går vi til steg 4, hvis det finnes går vi til steg 3.
3. Vi setter nå $work = work + Allocation_i$ og $finish[i] = true$ og går så til steg 2 igjen.
4. hvis $finish[i] = true$ for alle i er systemet i en sikker tilstand. Hvis man ikke kommet til steg 4 er systemet ikke i en sikker tilstand og i Bankers algoritmen kan dermed ikke ressursene tildeles før andre prosesser er fullført.

For å sjekke om en forespørsel kan godkjennes og tildeles må systemet sjekke at safety algoritmen etterlater systemet i en sikker tilstand dersom ressursene ble tildelt. Man sjekker da først om forespørselen er innenfor max grensene til prosessen og om det er nok tilgjengelige ressurser. Hvis det er det endrer man available og oppdaterer need og allocation for prosessen og sjekker da algoritmen over igjen med disse teoretiske nye preferansene. Hvis systemet med de nye preferansene fortsatt blir etterlatt i en sikker tilstand blir ressursene tildelt og systemets datastruktur oppdatert.

h)

i.

Tabellen, samt safety algoritmen, under viser at selskapet er i en sikker tilstand da rekkefølgen $\langle P2, P4, P3, P1 \rangle$ tilfredstiller safety conditions, altså at enhver prosess P_i har tilgjengelige ressurser for å fullføre sin prosess, evtl vente til P_j er fullført før den selv kan få ressursene den trenger og gjennomføre og avslutte sin prosess.

	Allokert			Max behov			Trenger (Need)			Sikker tilstand Rekkefølge	tid	Work/Available i tilstand t(i)		
	A	B	C	A	B	C	A	B	C			A	B	C
P1	0	1	0	7	5	3	7	4	3	t4	t0	2	3	0
P2	3	0	2	3	2	2	0	2	0	t1	t1	5	3	2
P3	3	0	2	9	0	2	6	0	0	t3	t2	6	4	4
P4	1	1	2	4	3	3	3	2	1	t2	t3	9	4	6
											t4	9	5	6

Med safety algoritmen får vi (steg 1-4 beskrevet i oppg. 1 g)

1. initialising: $work = 2\ 3\ 0$ og $finish[i] = false$ for ($i = 1, 2, 3, 4$)

2. $\text{finish}[2] = \text{false} \ \& \ \text{Need}_2 = 0 \ 2 \ 0 \leq 2 \ 3 \ 0 = \text{work}$
3. $\text{work} = \text{work} + \text{Allocated}_2 = 2 \ 3 \ 0 + 3 \ 0 \ 2 = 5 \ 3 \ 2 \ \& \ \text{finish}[2] = \text{true}$
2. $\text{finish}[4] = \text{false} \ \& \ \text{Need}_4 = 3 \ 2 \ 1 \leq 5 \ 3 \ 2 = \text{work}$
3. $\text{work} = 5 \ 3 \ 2 + 1 \ 1 \ 2 = 6 \ 4 \ 4$ og $\text{finish}[4] = \text{true}$
2. $\text{finish}[3] = \text{false} \ \& \ \text{Need}_3 = 6 \ 0 \ 0 \leq 6 \ 4 \ 4 = \text{work}$
3. $\text{work} = 6 \ 4 \ 4 + 3 \ 0 \ 2 = 9 \ 4 \ 6$ og $\text{finish}[3] = \text{true}$
2. $\text{finish}[1] = \text{false} \ \& \ \text{Need}_1 = 7 \ 4 \ 3 \leq 9 \ 4 \ 6 = \text{work}$
3. $\text{work} = 9 \ 4 \ 6 + 0 \ 1 \ 0 = 9 \ 5 \ 6$ og $\text{finish}[1] = \text{true}$
2. finner ingen $\text{finish} = \text{false} \rightarrow$ steg 4
4. $\text{finish}[i] = \text{true}$ for alle i , dermed er systemet i en *sikker tilstand*.

ii. Tabellen under viser tilstanden etter at forespørselen fra P4 er godkjent. Vi ser at allokeringen av (2,1,0) til P4 går fint da systemet forst tt kan f lge sekvensen <P2,P4,P3,P1> slik at safety conditions er oppfylt.

	Allokert			Max behov			Trenger (Need)			Sikker tilstand Rekkef�lge	tid	Work/Available i tilstand t(i)		
	A	B	C	A	B	C	A	B	C			A	B	C
P1	0	1	0	7	5	3	7	4	3	t4	t0	0	2	0
P2	3	0	2	3	2	2	0	2	0	t1	t1	3	2	2
P3	3	0	2	9	0	2	6	0	0	t3	t2	6	4	4
P4	3	2	2	4	3	3	1	1	1	t2	t3	9	4	6
											t4	9	5	6

iii. Tabellen under viser at tildelingen av (1,1,0) ikke er mulig for P1 da det ikke vil gi nok instanser i A til   fullf re alle prosesser. Verken P3 eller P1 vil kunne f  nok instanser fra A til   fullf re sine prosesser ved t3. Tildeingen kan dermed ikke godkjennes.

	Allokert			Max behov			Trenger (Need)			Sikker tilstand Rekkef�lge	tid	Work/Available i tilstand t(i)		
	A	B	C	A	B	C	A	B	C			A	B	C
P1	1	2	0	7	5	3	6	3	3		t0	1	2	0
P2	3	0	2	3	2	2	0	2	0	t1	t1	4	2	2
P3	3	0	2	9	0	2	6	0	0		t2	5	3	4
P4	1	1	2	4	3	3	3	2	1	t2	t3	ikke nok instanser i A		
											t4			

Oppg. 2

Se filen BoundedBuffer.java

Skjermskudd av kj ring:

```
BoundedBuffer
Produced one item to buffer.. size: 4
released lock, full size: 4
Produced one item to buffer.. size: 5
released lock, full size: 5
Consumed one item from buffer.. size: 4
released lock, full size: 4
Produced one item to buffer.. size: 5
released lock, full size: 5
Produced one item to buffer.. size: 6
released lock, full size: 6
Consumed one item from buffer.. size: 5
released lock, full size: 5
Produced one item to buffer.. size: 6
released lock, full size: 6
Produced one item to buffer.. size: 7
released lock, full size: 7
Produced one item to buffer.. size: 8
released lock, full size: 8
Produced one item to buffer.. size: 9
released lock, full size: 9
```

Oppg. 3

Se filen ReadersWriters.java

Skjermsskudd av kjøring:

```
ReadersWriters ReadersWriters ReadersWriters ReadersWriters ReadersWriters TheDiningPhilosophers ReadersWriters
Reading random number in database : 71
Reading first number in database : 71
Reading random number in database : 71
last reader access released..
Wrote 38 to database
first reader access acquired..
Reading last number in database : 38
Reading last number in database : 38
Reading first number in database : 71
Reading random number in database : 71
Reading first number in database : 71
Reading random number in database : 71
Reading last number in database : 38
Reading first number in database : 71
Reading random number in database : 71
last reader access released..
first reader access acquired..
Reading last number in database : 38
Reading last number in database : 38
Reading first number in database : 71
Reading last number in database : 38
Reading first number in database : 71
```

Oppg. 4

Se filen TheDiningPhilosophers.java

Skjermsskudd av kjøring:

```
ReadersWriters ReadersWriters ReadersWriters ReadersWriters ReadersWriters TheDiningPhilosophers TheDiningPhilosophers
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 2 is eating..
Philosopher 2 is thinking...
Philosopher 1 is thinking...
Philosopher 2 is thinking...
Philosopher 3 is thinking...
Philosopher 3 is eating..
Philosopher 0 is eating..
Philosopher 0 is thinking...
Philosopher 1 is eating..
Philosopher 4 is thinking
```

