5.4 Semaphores

Saturday, April 11, 2020 6:15 PM

Dette kapittelet ser nærmere på OS- og programmeringsspråk-mekanismer som brukes til å oppnå samtidighet, blant annet semaforer. Semaforer tilbyr en primitiv men kraftig og fleksibel måte å håndheve gjensidig utelukkelse og koordinering av prosesser.

Semaforer

En semafor er en heltallsverdi som brukes til signalisering mellom prosesser. Den generelle semaforen kan sees på som en variabel det kan utføres tre atomiske operasjoner på:

1. _initialisering: Semaforen settes til en ikke-negativ verdi.
2. _semWait: Dekrementerer semafor-verdien. Hvis semafor-verdien blir negativ, så vil prosessen som utfører _semWait-operasjonen blokkeres. Hvis ikke fortsetter prosessen execution.
3. _semSignal: Inkrementerer semafor-verdien. HVis den nye verdien er mindre enn eller lik null, så vil prosesser blokkert av _semWait bli ublokkert.

Hvis semafor-verdien er positiv vil det samme tallet tilsi hvor mange prosesser som kan sende ut et _semWait-kall og dermed starte execution med en gang. Dersom semafor-verdien er negativ tilsvarer tallet antallet prosesser som er blokkert.

Binær semafor

En binær semafor kan på selvforklarendevis kun besitte verdiene 0 eller 1. Den kan videre defineres ut fra de tre atomiske operasjonene:

1. _initialisering: En binær semafor-verdi kan initialiseres til 0 eller 1.
2. _semWaitB: Denne operasjonen sjekker semafor-verdien. Dersom verdien er 0 vil prosessen som utfører _semWaitB bli blokkert. Hvis verdien er 1 vil verdien endres til 0 og så starter execution.
3. _semSignalB: Denne operasjonen sjekker om en prosess er blokkert på denne semaforen (dvs at semafor-verdien er 0). Hvis den er er blokkert, så vil en prosess blokkert av _semwaitB bli ublokkert. Hvis ingen prosesser er blokkert så settes verdien til semaforen til 1.

Mutex-lås (gjensidig utelukkelse)

Mutex er et binært flagg som brukes til å låse og frigi et objekt. Når data som er opptatt ikke kan deles, eller at prosesseringen ikke kan foregå samtidig andre steder i systemet, så settes mutex til å låse seg (typisk ved 0), og blokkerer dermed andre forsøk på å bruke den. En mutex er satt til å låse seg opp igjen når dataen ikke brukes lengre eller rutinen er ferdig. Hovedforskjellen mellom en mutex og en binær semafor er at prosessen som låser mutex må være den samme som låser den opp.

Sterke semaforer bruker en FIFO-kø til å håndtere prosesser som veneter på semaforen.
Svake semaforer har ikke en spesifisert tilnærming for håndteringen av denne køen.

Gjensidig utelukkelse ved bruk av semaforer

Figuren under demonstrerer implementasjonen. Vi ser på *n* prosesser, identifisert i listen *P(i)* som kan tenkes på som en kø som holder på alle prosesser som skal aksessere samme ressurs. Hver prosess har en kritisk seksjon som bruker denne ressursen. For hver prosess kjøres det en _semWait(s) rett før dens kritiske seksjon. Dersom verdien på *s* blir negativ vil prosessen blokkeres. Dersom verdien er *s = 1* så vil den dekrementeres til 0, og prosessen kjører da sin kritiske seksjon umiddelbart. Ettersom *s* ikke lenger er positiv vil ingen andre prosesser kunne entre sin kritiske seksjon.

```
/*  program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int  i)
{
     while (true) {
          semWait(s);
          /* critical section */;
          semSignal(s);
          /* remainder */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . . , P(n));
}
```

Figure 5.9 Mutual Exclusion Using Semaphores

I utgangspunktet initialiseres semafor-verdien til 1. For å støtte kravet om at flere prosesser skal kunne entre sin kritiske seksjon om gangen kan man andre spesifiseringen av initial-verdien til semaforen. Dermed kan man tolke verdien av *s.count* som følgende:

- *s.count* ≥ 0: I dette tilfellet er *s.count* antall prosesser som kan execute _semWait(s) uten suspendering (hvis ingen _semSignal(s) utføres i mellomtiden). Slike situasjoner vil tillate semaforer å støtte synkronisering og gjensidig utelukkelse
- *s.count* < 0: Størrelsen på *s.count* vil I dette tilfellet tilsvare antall prosesser som er suspendert (er i hvilemodus) i *s.queue*

Figuren under demonstrerer prosesser som aksesserer delte dataer beskyttet av en semafor. Merk at normale operasjoner kan kjøres i parallell, mens kritiske seksjoner må serialiseres.

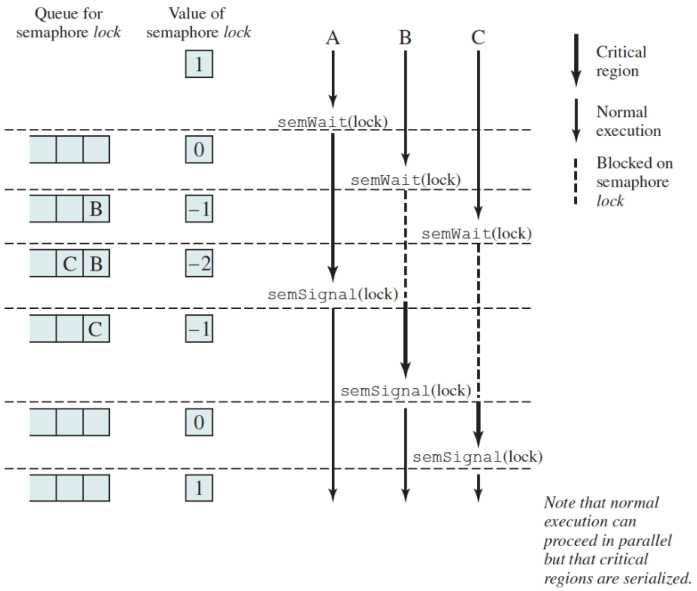

Figure 5.10 **Processes Accessing Shared Data Protected by a Semaphore**

The Producer/Consumer Problem

Dette er et av de vanligste problemene som oppstår ved samtidig prosessering. Problemet omfatter en *producer* som genererer ulike typer data (rader, tegn) til buffer, og en *consumer* som tar dataen ut av bufferet for å benytte det. Disse to partene kommuniserer via en kø med maksimal størrelse *N* og har følgende begrensninger:

1. Kun én bufferoperasjon kan foregå om gangen
2. Consumer må vente på producer dersom køen er tom
3. Producer må vente på consumer dersom køen er full

De to siste av de ovennevnte punktene omhandler synkronisering.

Løsning på problemet med semaforer

Dette problemet kan løses ved hjelp av semaforer. La semaforen *e* være antall tomme plasser i køen. La *n* være antall elementer i køen. Semaforen *s* ivaretar integriteten til køens tilstand, for eksempel hvis to producers forsøker å legge til objekter til en tom kø samtidig.

Figuren under er en implementasjon av denne løsningen.

```
/* program boundedbuffer */
      const int sizeofbuffer = /* buffer size */;
      semaphore s = 1, n = 0, e = sizeofbuffer;
      void producer()
      {
           while (true) {
                 produce();
                 semWait(e);
                 semWait(s);
                 append();
                 semSignal(s);
                 semSignal(n);
           }
      }
      void consumer()
      {
           while  (true) {
                 semWait(n);
                 semWait(s);
                 take();
                 semSignal(s);
                 semSignal(e);
                 consume();
           }
      }
      void main()
      {
           parbegin  (producer, consumer);
      }
```

Figure 5.16 **A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores**

Eksempelkoden forklart ytterlige:

1. En consumer går inn i sin kritiske seksjon. Siden *n = 0* blokkeres consumeren.
2. Flere producers går inn i sin kritiske seksjon. Maksimalt *sizeofbuffer* kan produseres.
3. Det er kun én producer som får aksess til bufferet om gangen og legger til data i køen. Dette håndheves av semaforen *s.*
4. Idet produceren går ut av sin kritiske seksjon økes *n*, hvorpå et signal så sendes slik at en consumer kan gå inn i sin kritiske seksjon.

Ved semaforer ligger ansvaret for gjensidig utelukkelse og synkronisering hos programmerer.