5.6 Message passing

Friday, April 17, 2020 1:14 AM

Når to prosesser samhandler med hverandre er det to fundamentale krav som må være på plass: synkronisering og kommunikasjon. Prosesser må synkroniseres for å håndheve gjensidig utelukkelse; prosesser som samarbeider må kanskje utveksle informasjon. Meldingsutveksling (*message passing*) er en løsning på dette tilfellet. Det kan implementeres både i distribuerte systemer og delt-minne systemer som multiprosessor- og uniprosessor-systemer.

Funksjonene som muliggjør meldingsutveksling er et sett med to primitiver. Dette er det minimale settet med operasjoner nødvendig for meldingsutveksling:

- send(destination, message)
- receive(source, message)

Første argument i funksjonene indikerer hvilken prosess som er henholdsvis mottaker og avsender. Selve informasjonen som sendes er i form av message-argumentet.

Synkronisering

Sender-prosessen og mottaker-prosessen kan enten være blokkerende eller ikke-blokkerende.

1. **Blokkerende avsender, blokkerende mottaker:** Både avsender og mottaker er blokkert inntil en melding er levert. Denne kombinasjon kalles gjerne *rendezvous*, og er en kombinasjon som tillater streng synkronisering mellom prosesser.
2. **Ikke-blokkerende avsender, blokkerende mottaker:** Mottaker blokkeres inntil meldingen er mottatt. Dette er kanskje den mest hensiktsmessige kombinasjonen da den tillater en prosess å sende en eller flere meldinger til flere ulike destinasjoner så fort som det lar seg gjøre. En prosess som må motta en melding før den kan gjøre nyttig arbeid må blokkeres inntil den får den tilsendte meldingen. Et eksempel er en server-prosess som eksisterer for å tilby en tjeneste eller ressurs til andre prosesser.
3. **Ikke-blokkerende avsender, ikke-blokkerende mottaker:** Hverken avsender eller mottaker er påkrevd å vente.

Den ikke-blokkerende avsenderen brukes vanligvis i mange samtidsprogrammerings-oppgaver. Det kan i verste tilfelle oppstå overbruk av systemressurser som prosessortid og bufferplass ved bruk av ikke-blokkerende avsender. Dette skyldes at en error kan resultere i at prosessen sender gjentatte genererte meldinger. Bekreftelse på mottagelse av meldinger hos mottaker må implementeres av utvikleren ved bruk av ikke-blokkerende avsender.

For mottaker-primitiven er det vanligere med den blokkerte versjonen i samtidsprogrammerings-oppgaver. Utfordringen ved denne versjonen er at dersom en melding er tapt, så risikerer mottaker å forbli blokkert. Den ikke-blokkerende varianten har ikke dette problemet, men problemet med denne ikke-blokkerende varianten er at hvis en melding sendes etter at en prosess allerede har executed en matchende mottaker, så vil meldingen gå tapt.

Adressering

Adressering handler om å spesifisere hvilke prosesser som skal motta en melding fra avsender-primitiven (_send). Det er to typer adressering:

- **Direkte adressering:** Avsender inkluderer en spesifikk identifikator til destinasjonsprosessen.
- **Indirekte adressering:** Meldinger sendes via en delt datastruktur som består av køer som midlertidig holder på meldingene. Disse køene kalles gjerne mailboks (*mailboxes*). I dette tilfelle kommuniserer prosessene ved å sende meldinger til mailboksen, slik at andre prosesser kan hente ut meldingen. Denne uavhengige ordningen mellom avsender og mottaker gir økt fleksibilitet.

Forholdet mellom avsender og mottaker har flere type relasjoner:

- **1:1 –** Tilsier privat kommunikasjon mellom to prosesser.
- **M:1 –** Tilser at mange prosesser kan kommunisere med én bestemt prosess. Dette er typisk hensiktsmessig for klient/server-tjenester. I dette tilfellet er da mailbox server og kalt *port*.
- **1:M –** Tilsier at en avsender kan ha flere mottakere. Dette er hensiktsmessig for applikasjoner hvor en melding eller informasjon skal kringkastes (*broadcast*) til et sett med prosesser.
- **M:M –** Tilsier at flere server-prosesser kan tilby samtidig tjeneste til flere klienter.

Forholdet mellom prosesser og mailbokser kan enten være statisk eller dynamisk.

- **Statisk:** I dette tilfellet er en prosess permanent tilkoblet en bestemt bort. (1:1)
- **Dynamisk:** I tilfellet hvor det er mange avsendere kan forholdet til mailboks oppstå dynamisk.

Primitivene _connect og _disconnect kan typisk brukes til ovennevnt bruk av dynamisk/statisk tilkobling.

Figuren under viser det typiske formatet på en melding.

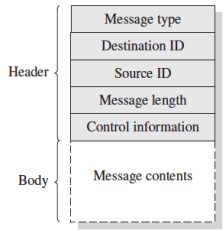

Figure 5.22 General Message Format

Gjensidig utelukkelse med meldinger

```
/* program mutualexclusion */
const int n = /* number of process */
void P(int i)
{
      message msg;
      while (true) {
         receive (box, msg);
         /* critical section */;
         send (box, msg);
         /* remainder */;
      }
}
void main()
{
      create mailbox (box);
      send (box, null);
      parbegin (P(1), P(2), . . . , P(n));
}
```

Figure 5.23 Mutual Exclusion Using Messages

Kodesnutten over er en eksempelkode på implementasjon av gjensidig utelukkelse med meldinger. Vi antar ikke-blokkerende avsender, og blokkerende mottaker som primitiver. Et sett med parallelle prosesser deler mailboks, kalt *box*, som brukes av alle prosesser til sending og mottagelse av data. Mailboksen inneholder en melding med tomt innhold fra starten.

1. En prosess som ønsker å gå inn i sin kritiske seksjon forsøker først å motta en melding fra mailboks.
2. Dersom mailboksen er tom blokkeres denne prosessen. Hvis ikke, så mottar prosessen meldingen og går inn i sin kritiske seksjon. Meldingen plasseres tilbake i mailboksen box.

   Merk at den tomme meldingen altså er en *token* prosessen må besitte for å kunne gå inn i sin kritiske seksjon.

Denne modellen antar at dersom en eller flere prosesser utfører mottaker-operasjonen samtidig så:

- Hvis det er en melding, så leveres den kun til én av prosessene, mens den andre blokkeres.
- HVis meldingskøen er tom, så blokkeres alle prosessene; når en melding blir tilgjengelig aktiveres kun én av prosessene for å motta meldingen.

The Producer/Consumer problem (med meldingsutveksling)

```
const int
    capacity = /* buffering capacity */ ;
    null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
     receive (mayproduce,pmsg);
     pmsg = produce();
     send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
     receive (mayconsume,cmsg);
     consume (cmsg);
     send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

Figure 5.24 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

Kodesnutten over er en løsning på bounded-buffer producer/consumer-problemet. Her sendes data og signaler og data som meldinger. Det er to mailbokser; *mayconsume* som inneholder data generert av producer i form av meldinger, og *mayproduce* som inneholder et antall null-meldinger lik kapasiteten av bufferet (mayconsume). Mayconsume-mailboksen har rollen som buffer; organisert som en meldingskø med en gitt størrelse lik den globale variabelen *capacity*. Så lenge det er en melding i mayconsume kan en consumer forbruke. Mayproduce reduseres i antall meldinger ved hver produksjon av producer, og øker ved hvert forbruk av consumer.

Denne metoden er ganske fleksibel. Det kan være flere producers og consumers så lenge alle har aksess til begge mailbokser. Løsningen fungerer også for distribuerte systemer der mayproduce kan ligge et sted, og mayconsume et annet sted.