5.5 Monitors

Wednesday, April 15, 2020 2:18 PM

En monitor er en del av et programmeringsspråk som innkapsler variabler, aksesserer prosedyrer og initierer kode innenfor en abstrakt datatype. Det ligner dermed på en utvidelse av klassekonseptet. Variablene til monitoren kan kun aksesseseres gjennom bestemte aksessprosedyrer, og kun én prosess kan aksessere monitoren om gangen. Aksessprosedyrene er kritiske seksjoner, og en monitor kan håndtere en kø med prosesser som venter på aksess. Monitorer er altså mer høy-nivå enn semaforer, noe som gjør det lettere å verifisere korrektheten.

Monitorer med signal

Monitorer er en softwaremodul bestående av en eller flere aksessprosedyrer, en initialiseringssekvens og lokal data. Egenskaper hos en monitor:

1. De lokale datavariablene er kun aksesserbare av monitorens prosedyrer og ikke av en ekstern prosedyre.
2. En prosess entrer monitoren ved å påkalle en av dens prosedyrer.
3. Kun én prosess kan kjøre i monitoren om gangen; alle andre prosesser som har påkallet monitoren blir blokkert, og må vente til monitoren blir tilgjengelig. Dette impliserer gjensidig utelukkelse.

De to første ovennevnte punktene minner om objekt-strukturen og kan dermed enkelt implementeres i et OOP ved å legge til noen ekstra egenskaper. Figuren under viser strukturen til en monitor.

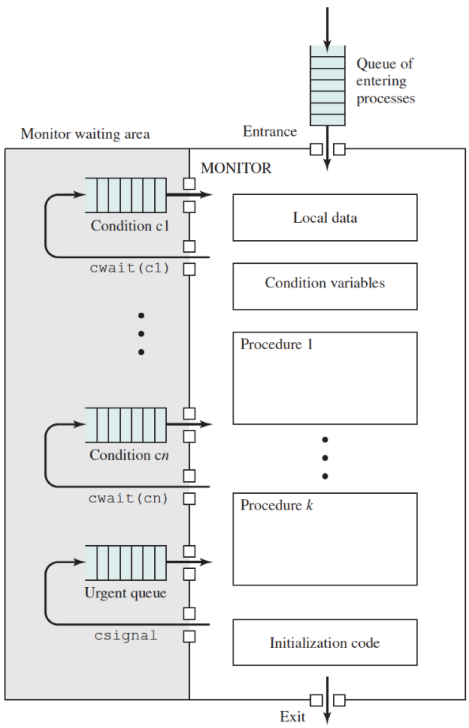

Figure 5.18 Structure of a Monitor

Monitoren støtter synkronisering ved bruk av *betingelsesvariabler* (også kalt tilstandsvariabel). Disse ligger i monitoren og er kun aksesserbare i monitoren. Betingelsesvariabler er en spesiell datatype i monitorer, som opereres ved to funksjoner:

- _cwait(c): Suspender den kallende prosessen på betingelse *c*. Dette tilgjengeliggjør monitoren for andre prosesser.
- _csignal(c): Fortsetter execution av en prosess blokkert av _cwait på den samme betingelsen *c*. Hvis det er flere slike prosesser så velges én av dem; hvis det ikke er noen slike prosesser gjøres det ingenting.

Merk at *wait*- og *signal*-operasjonene for en monitor er annerledes enn de for semaforer. Hvis en prosess i en monitor ufører en signal-operasjon og det ikke er noen oppgaver som venter på betingelsesvariabelen så blir signalet tapt/borte.

Prosesser som venter på tilgang til monitoren plasseres i en kø av blokkerte prosesser som venter på aksess. Når en prosess er i monitoren kan den imidlertid blokkere seg selv på betingelsen *x*; ved funksjonskallet _cwait(x). Denne prosessen plasseres da i køen som venter på gjentatt tilgang til monitoren når betingelsen endres, og fortsette på execution av programmet sitt påfulgt av _cwait(x)-kallet.

Hvis en prosess som executer i monitoren oppdager en endring i betingelsesvariabelen *x*, så kan den bruke funksjonskallet _csignal(x) som da informerer den tilhørende betingelseskøen at betingelsen har endret seg.

Ulemper (monitor med signal)

Denne modellen krever at dersom det hvert fall er én prosess i betingelseskøen, så må en prosess fra den køen kjøre umiddelbart etter en annen prosess har kalt _csignal(c) for betingelsen *c*. Altså må prosessen som kalte funksjonen _csignal() umiddelbart frigjøre monitoren eller bli blokkert på monitoren. Dette gjør at det er to ulemper til denne modellen:

1. Hvis prosessen som kalte _csignal() ikke har blitt ferdig med monitoren, så kreves det to andre brytere: én til å blokkere prosessen, og en annen til å fortsette når monitoren blir tilgjengelig.
2. Prosessplanleggingen tilknyttet et signal må være fullstendig pålitelig. Ved et kall av _csignal() må en prosess fra den tilhørende betingelseskøen aktiveres umiddelbart, og planleggeren må sikre at ingen andre prosesser entrer monitoren før aktivering. Hvis ikke kan betingelsen som aktiverte prosessen endres.

The Producer/Consumer problem (ved bruk av monitor)

Vi har igjen et *bounded buffer*. I dette tilfelle vil det si en monitormodul som kontrollerer bufferet som brukes til å lagre og hente ut data. Monitoren har to betingelsesvariabler av type *cond*: den første er *notfull* som er True dersom det er plass til å legge til et nytt element i bufferet; den andre er *notempty* som er True dersom det er hvert fall ett element i bufferet.

En producer kan legge til data i bufferet ved å bruke *append*-funksjonen til monitoren, altså har den ikke direkte aksess til bufferet og må gjennom monitoren. Denne funksjonen sjekker først betingelsesvariabelen *notfull* for å avgjøre om det er nok plass i bufferet. Hvis det ikke er nok plass vil prosessen blokkeres på den betingelsen. I så fall kan en annen prosess (producer eller consumer) oppta monitoren. Den blokkerte prosessen kan fjernes fra køen når det er ledig plass i bufferet og dermed fortsette sin execution. Etter en prosess har plassert sitt element i bufferet vil den signalisere *nonempty*-variabelen. Merk den gjensidige utelukkelsen ved at producer og consumer ikke kan aksessere bufferet samtidig.

Figuren under viser eksempelkode på producer/consumer:

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                        /* space for N items */
int nextin, nextout;                                    /* buffer pointers */
int count;                                          /* number of items in buffer */
cond notfull, notempty;              /* condition variables for synchronization */
void append (char x)

{
     if (count == N) cwait(notfull);          /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal (notempty);                          /*resume any waiting consumer */
}
void take (char x)
{
     if (count == 0) cwait(notempty);       /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N);
     count--;                                      /* one fewer item in buffer */
     csignal (notfull);                          /* resume any waiting producer */
}
{                                                                /* monitor body */
     nextin = 0; nextout = 0; count = 0;             /* buffer initially empty */
}
```

```
void producer()
{
     char x;
     while (true) {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true) {
     take(x);
     consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.19 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Monitor with Notify and Broadcast (MESA)

Denne modellen løser problemet med to ulempene fra monitor med signal. I denne strukturen er _csignal(x) byttet ut med funksjonen _cnotify(x). Funksjonen _cnotify(x) vekker betingelseskøen til *x*, men den signaliserende prosessen fortsetter å execute. Resultatet av at køen vekkes er at prosessen i toppen av betingelseskøen vil fortsette på et bekvemmelig tidspunkt når monitoren blir tilgjengeliggjort. Ettersom det ikke er noen garanti for at en annen prosess ikke får entre monitoren før den ventende prosessen så må den ventende prosessen sjekke betingelsen på nytt. Ved denne implementasjonen blir funksjonene for *bounded-buffer* annerledes; merk at if-setningene er erstattet av en while-løkke. Dette gjør at det er en ekstra evaluering av betingelsesvariabelen, samt ingen ekstra prosessbytter eller restriksjoner når den ventende prosessen skal kjøre etter _cnotify().

```
void append (char x)
{
      while (count == N) cwait(notfull);      /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;                                    /* one more item in buffer */
      cnotify(notempty);                        /* notify any waiting consumer */
}
void take (char x)
{
      while (count == 0) cwait(notempty);   /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                    /* one fewer item in buffer */
      cnotify(notfull);                         /* notify any waiting producer */
}
```

Figure 5.20 Bounded-Buffer Monitor Code for Mesa Monitor

En timer brukes til å sette et maksimalt tidsintervall for hvor lenge en prosess kan vente på monitoren. Dette forhindrer starvation som kan oppstå når en prosess feiler før den får signalisert betingelsen gjennom _cnotify. En prosess som har ventet tilsvarende det maksimale tidsintervallet vil aktiveres uavhengig av om den er blitt signalisert etter timeout. Denne prosessen sjekker da betingelsesvariabelen og fortsetter kjøringen dersom den er tilfredsstilt.

Denne ideen om at en prosess blir varslet istedenfor direkte reaktivert gjør at det er mulig å legge til en primitiv i form av _cbroadcast(). Denne operasjonen gjør at alle prosesser som venter på en betingelse blir plassert i Ready. Dette er hensiktsmessig i situasjoner hvor en prosess ikke vet hvor mange andre prosesser som bør reaktiveres.

Fordeler MESA

Denne metoden har færre error enn monitor med signal. Dette skyldes at den signaliserende prosessen sjekker betingelsen ved while-løkken. En prosess kan signalisere feil uten å skape problemer siden betingelsen sjekkes før kjøringen fortsetter. Denne metoden er i tillegg mer modulær da MESA har fasiliteter for interprosedyriske modularitetsprinsipper.

Monitor/signal	**Monitor/MESA**
Aktøren som vekker opp andre prosesser gjennom _csignal vil midlertidig frigjøre monitoren slik at den påkallede aktøren (den som ble vekket opp) som tidligere benyttet seg av _cwait kan fortsette uten å måtte teste for betingelsen på nytt. Dette manglende behovet for retesting av betingelser er billig for applikasjonene, men de mange resulterende tråd- og prosessorskiftene er dyrt for systemet.	Aktøren som vekker opp andre prosesser gjennom _cnotify vil ikke frigjøre monitoren for den som tidligere har kalt _cwait. Retesting av systemet er dyrt, men de få resulterende tråd- og prosessorskiftene er billig for systemet. I tillegg har MESA en broadcast-funksjon hvor en aktør kan vekke opp alle aktører som venter på betingelsen av gangen. Dette er eksempelvis nyttig når en memory manager vil sjekke hvor mye tilgjenglig minne den har, og dermed bare kan vekke opp prosessene så de kan sjekke selv.