5.3 Mutual exlusion: hardware support

Saturday, April 11, 2020 2:14 AM

Avbruddsdeaktivering (interrupt disabling)

I et uniprosessor-system, så kan ikke samtidige prosesser ha overlappende execution; de kan kun innflettes. I tillegg vil en prosess fortsette å kjøre til den kaller på en tjeneste fra OS-et eller inntil den blir avbrutt. Det er derfor nødvendig å forhindre at en prosess kan bli avbrutt for å sikre gjensidig utelukkelse. OS-kjernen kan definere primitiver for å deaktivere og aktivere avbrudd. Det gjøres på følgende måte (håndheving av gjensidig utelukkelse):

```
while (true) {
   /* disable interrupts */;
   /* critical section */;
   /* enable interrupts */;
   /* remainder */;
}
```

Fordi en kritisk seksjon ikke kan bli avbrutt, så er gjensidig utelukkelse garantert. Denne metoden har høy kostnad fordi kjernen ikke kan flette prosesser. Denne metoden sikrer ikke gjensidig utelukkelse for multiprosessorsystemer, ettersom flere prosesser kan kjøre samtidig.

Spesielle maskin-instruksjoner

På hardware-nivå, vil aksess til en minnelokasjon ekskludere enhver annen aksess til samme minnelokasjon. Dette danner grunnlagt for spesielle maskininstruksjoner som utfører to handlinger atomisk, enten ved read/write eller read/test. Ved utførelse av instruksjonen, blokkeres tilgangen til minnelokasjonen for enhver annen instruksjon som refererer til samme lokasjon.

Fordelen med å forhindre avbrudd er at det både fungerer for uniprosessorsystemer og multiprosessorsystemer som deler samme hovedminne. Metoden er attpåtil enkel, som gjør den lett å verifisere. Den støtter flere kritiske seksjoner ettersom hver kritiske seksjon kan defineres som en egen variabel.

Ulempene med å forhindre avbrudd er at en prosess blir satt til busy waiting når den venter på tilgang til den kristiske seksjonen, slik at den fortsetter å oppta prosessortid. I tillegg kan det oppstå starvation. Ved prioriteringsnivåer kan det oppstå vranglåser.

Compare&swap

CAP er en atomisk instruksjon over to steg: en sammenligning av en minneverdi og en testverdi, hvis verdiene er de samme oppstår det et bytte (testval -> newval). Metoden bruker flertråding for å oppnå synkronisering. Atomisiteten sikrer at verdiene beregnes fra den siste oppdaterte informasjonen. Hvis verdien har blitt oppdatert av en annen tråd i mellomtiden vil nemlig write-operasjonen feile. Minneverdien er en adresse (*word) og testverdien (testval) er også en adresse.

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

Exchange instruction

I denne instruksjonen byttes innholdet i et register med innholdet i en minneadresse.

```
void exchange (int *register,  int *memory)
{
   int temp;
   temp = *memory;
   *memory = *register;
   *register = temp;
}
```