Sünkroniseerimine

Ülevaade

- Taust
- Kriitilise sektsiooni probleem
- Sünkroniseerimisriistvara
- Semaforid
- Klassikalised sünkroniseerimisprobleemid
- Kriitilised regioonid
- Monitorid
- Näited: Solaris, Windows 2000

Taust

- Paralleelne juurdepääs jagatud mälule võib andmete kooskõla rikkuda
- Andmete konsistentsena hoidmiseks on vaja mehhanismi, et koostööd tegevaid protsesse õiges järjekorras täita
- Jagatud mälu puhver $n-1$ kirjega oli lihtne, n kirjega keerulisem
  - Näiteks lisame juurde loenduri (näitab, mitu elementi puhvris on, algul 0)

Näide: seesama piiratud puhver

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Näide: piiratud puhver, tootja ja tarbija

- Tootja protsess

```
item nextProduced;
while (1) {
    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```


OPERATSIOONISÜSTEEMIDE EHITUS

Näide: piiratud puhver, tootja ja tarbija

- Tarbija protsess

```
item nextConsumed;
while (1) {
    while (counter == 0)
        ; /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```


OPERATSIOONISÜSTEEMIDE EHITUS

Atomaarsed operatsioonid

- `counter++` ja `counter—` tuleb täita atomaarselt
- Atomaarne operatsioon on operatsioon, mis täidetakse ühekorraga ilma katkestamata
- Lause `counter++` ja `counter—` võivad kumbki olla realiseeritud kolme masinkäsuna:

```
reg1 = counter          reg2 = counter
reg1 = reg1 + 1         reg2 = reg2 — 1
counter = reg1          counter = reg2
```

- Need käsujadad erinevates protsessides võivad üksteisega vahelduda kasvõi ühe käsu kaupa
- Mis on tulemus, kui `counter = 5`, üks protsess teeb `counter++` ja teine samal ajal `counter—` — kas 4, 5 või 6?

OPERATSIOONISÜSTEEMIDE EHITUS

Võidujooks ja kriitiline sektsioon

- Võidujooks (*race condition*) — situatsioon, kus andmete kasutamisel mitme protsessi poolt korraga sõltub tulemus ajalistest teguritest
- Võidujooksude vastu aitab protsesside sünkroniseerimine
- Olgu meil n protsessi võistlemas samade jagatud andmete pärast
- Iga protsessi koodisegmendist mingi jupp on kriitiline — see jupp, kus tegeldakse jagatud andmetega
- Probleem: kindlustada, et kui üks protsess täidab oma kriitilist sektsiooni, siis teised protsessid ei saa oma kriitilisi sektsioone täita

Kriitilise sektsiooni probleemi lahendus

- Vastastikune välistamine — kui protsess P täidab oma kriitilist sektsiooni, siis ükski teine protsess ei tohi oma kriitilist sektsiooni täita
- Progress — kui ükski protsess ei täida oma kriitilist sektsiooni ja leidub mingi protsess, mis soovib siseneda kriitilisse sektsiooni, siis kriitilisse sektsiooni siseneva protsessi valikut ei saa lõpmatuseni edasi lükata
- Piiratud ootamine — peab leiduma ülempiir sellel kordade arvul, mitu korda muud protsessid sisenevad kriitilisse sektsiooni sel ajal, kui protsess on avaldanud soovi kriitilisse sektsiooni siseneda ning seda soovi pole veel täidetud
  - Eeldatakse, et iga protsess edeneb nullist suurema kiirusega
  - Protsesside omavahelise suhtelise kiiruse kohta eeldusi ei tehta

Lahenduse kandidaate

- Eeldame ainult 2 protsessi P_0 ja P_1
- Protsessi P_i üldine skeem (teine protsess on P_j)

```
do {
    sisenemise sektsioon
    kriitline sektsioon
    väljumise sektsioon
    ülejäänud sektsioon
} while (1);
```


Kandidaat-algoritm 1

- Jagatud muutujad:
  `int turn = 0` (näitab, mitmes protsess võib kriitilisse sektsiooni minna)
- Protsess P_i:

```
do {
    while (turn != i) ;
    critical section
    turn = j;
    reminder section
} while (1);
```

- Rahuldab vastastikuse välistamise tingimust, aga mitte progressi tingimust

Kandidaat-algoritm 2

- Jagatud muutujad:
  `boolean flag[2];` (algselt `flag[0] = flag[1] = false`)
- `flag[i] = true` $\Rightarrow$ P_i on valmis sisenema kriitilisse sektsiooni
- Protsess P_i:

```
do {
    flag[i] = true;
    while (flag[j]) ;
    critical section
    flag [i] = false;
    reminder section
} while (1);
```

- Rahuldab vastastikuse välistamise tingimust, aga mitte progressi tingimust

Kandidaat-algoritm 3

- Jagatud muutujad: mõlema eelmise algoritmi omad
- Protsess P_i:

```
do {
    flag [i] = true;
    turn = j;
    while (flag [j] and turn = j) ;
    critical section
    flag [i] = false;
    reminder section
} while (1);
```

- Rahuldab kõiki meie tingimusi

Pagari algoritm (1)

- Kriitiline sektsioon n protsessi jaoks
- Enne kriitilisse sektsiooni sisenemist saab iga protsess numbri
- Vähima numbri saanu siseneb kriitilisse sektsiooni
- Võib juhtuda, et protsessid P_i ja P_j saavad sama numbri
- Sel juhul teenindatakse protsessi, mille nimi (i või j) on väiksem
- See skeem garanteerib alati monotoonselt mittekahaneva järjestuse: näiteks 1,2,3,3,3,3,4,5...
- Jagatud muutujad:

`boolean choosing[n];` (initsialiseeritakse `choosing[k] = false`)
`int number[n];` (initsialiseeritakse `number[k] = 0`)

Pagari algoritm (2)

- Protsess P_i:

```
do {
  choosing[i] = true;
  number[i] = max(number[0], number[1], ..., number [n-1])+1;
  choosing[i] = false;
  for (j = 0; j < n; j++) {
    while (choosing[j]) ;
    while ((number[j] != 0) && ((number[j],j) < (number[i],i)));
  }
  critical section
  number[i] = 0;
  remainder section
} while (1);
```


Sünkroniseerimisriistvara

- Atomaarne kontroll + omistamine:

```
boolean TestAndSet(boolean &new) {
    boolean rv = new;
    new = true;
    return rv;
}
```

- Atomaatne vahetamine:

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Atomaarse operatsiooni kasutamine

- Jagatud muutujad:
  `boolean lock = false;`
- Protsess P_i:

```
do {
    while (TestAndSet(lock)) ;
    critical section
    lock = false;
    reminder section
} while (1);
```


Semaforid

- Sünkroniseerimisvahend, mis ei vaja kogu ooteajaks hõivatud ootamist (*busy wait*)
- Semafor S — täisarvuline muutuja
- Ainsaks kasutusviisiks kaks atomaarset operatsiooni:
  - Semafori võtmine (vajadusel ootamisega):

```
wait(S) {
    while (S<=0) do ;
    S--;
}
```

  - Semafori lahti laskmine:

```
signal (S) {
    S++;
}
```


Kriitiline sektsioon mitme protsessiga

- Jagatud muutujad:
  `semaphore mutex = 1`
- Protsess P_i:

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```


Semafor kui üldine sünkroniseermisvahend

- Tahame, et operatsioon B protsessis P_j täidetaks alles pärast operatsiooni A täitmist protsessis P_i
- Kasutame semafori (flag), initsialiseerime ta 0-ks
- Pseudokood:

P_i	P_j
...	...
A	wait(flag)
signal(flag)	B

Semafori realiseerimine (1)

- Defineerime semafori kui struktuuri

```
typedef struct {
   int value;
   struct process *L;
} semaphore;
```

- Kasutame kahte lihtsat operatsiooni:
  - `block()` blokeerib seda kasutava protsessi
  - `wakeup(P)` jätkab blokeeritud protsessi P

Semafori realiseerimine (2)

- Defineerime semafori operatsioonid nüüd nii:

```
wait(S) {
  S.value-;
  if (S.value < 0) {
    add this process to S.L;
    block();
  }
}
```

```
signal(S) {
  S.value++;
  if (S.value <= 0) {
    remove a process P from S.L;
    wakeup(P);
  }
}
```

- Vajame endiselt atomaarsust
- Atomaarsuse saavutame lühikese hõivatud ootamisega
- Ise hõivatud ootamist ei kasuta

Tupikud ja näljutamine

- Tupik — kaks või enam protsessi ootavad lõpmatul sündmust, mida saaks põhjustada ainult mingi teine ootav protsess
- Olgu S ja Q kaks semafori (algväärtusega 1)

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);

- Näljutamine — lõpmatu blokeerimine, kus mingi protsess jääb igaveseks semafori järjekorda (näiteks LIFO järjekord ja suur koormus)

Semaforide tüübid

- Loendav semafor — täisarv võib suvalisi väärtusi võtta
- Binaarne semafor — täisarv võib olla ainult 0 või 1 (kohati lihtsamini realiseeritav)
- Loendavat semafori saab realiseerida binaarsete semaforide abil:
  - Andmestruktuurid:

```
binary-semaphore S1 = 1, S2 = 0;
int C;
```

  - C algväärtuseks olgu S algväärtus

Loendav semafor binaarsetest semaforidest

- wait(S):

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

- signal(S):

```
wait(S1);
C++;
if (C <= 0)
    signal(S2);
else
    signal(S1);
```


Klassikalised sünkroniseerimisprobleemid

- Piiratud puhver
  - Fikseeritud pikkusega massiiv elementidest, tootja ja tarbija
- Lugejad ja kirjutajad
  - Üks (atomaarselt mitte kasutatav) andmestruktuur
  - Lugejad ja kirjutajad
  - Mitu lugejat saab korraga olla, kirjutaja tahab üksi olla
- Einestavad filosoofid
  - Palju ressursse, palju kasutajaid, iga kasutaja kasutab mitut ressurssi

Piiratud puhver semaforidega (1)

- Jagatud muutujad:

```
semaphore full=0, empty=n, mutex=1;
```

- Tootja:

```
do {
  ...
  produce an item in nextp
  ...
  wait(empty);
  wait(mutex);
  ...
  add nextp to buffer
  ...
  signal(mutex);
  signal(full);
} while (1);
```


Piiratud puhver semaforidega (2)

- Tarbija protsess:

```
do {
  wait(full)
  wait(mutex);
  ...
  take an item from buffer to nextc
  ...
  signal(mutex);
  signal(empty);
  ...
  consume the item in nextc
  ...
} while (1);
```

Lugejad-kirjutajad (1)

- Jagatud muutujad:

```
semaphore mutex=1, wrt=1;
int readcount=0;
```

- Kirjutaja:

```
    wait(wrt);
    ...
    writing is performed
    ...
    signal(wrt);
```

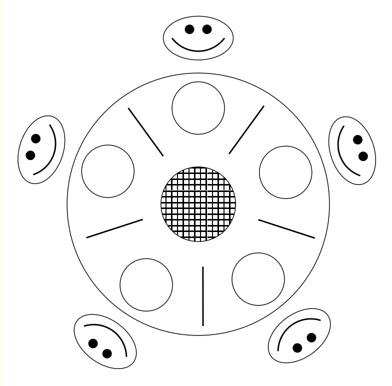

Lugejad-kirjutajad (2)

- Lugeja:

```
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(rt);
    signal(mutex);
    ...
    reading is performed
    ...
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
```


Einestavad filosoofid

- 5 filosoofi ümber laua
- Iga filosoof mõtleb ja sööb vaheldumisi
- Riisi söömiseks vaja pulki kummaltki poolt



Filosoofide algoritm

- Jagatud andmed:
  `semaphore chopstick[5];` (algselt 1)
- Filosoof P_i:

```
do {
  wait(chopstick[i])
  wait(chopstick[(i+1) % 5])
  ...
  eat
  ...
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
  ...
  think
  ...
} while (1);
```

Kriitilised regioonid

- Semaforide kasutamine on keerukas — mis saab, kui programmeerija kasutab wait+signal asemel signal+wait, wait+wait, signal+signal kogemata või unustab ühe üldse ära?
- Kriitiline regioon — kõrgema taseme sünkroniseerimiskonstruktsioon (peaks olema lollikindlam)
- Jagatud muutuja defineerimine:
  v: `shared` T
- Muutujat v kasutatakse ainult konstruktsioonis
  `region` v `when` B `do` S (kus B on tõeväärtusavaldis)
- Lause S täitmise ajal ei saa teised protsessid kasutada muutujat v
- Täitmine blokeerub lisaks ka seni, kuni avaldis B täidetuks saab
- Näide: Java `synchronized()`

Näide: puhver kriitiliste regioonidega (1)

- Jagatud andmed:

```
struct buffer {
    int pool[n];
    int count, in, out;
}
```

- Tootja:

```
region buffer when (count < n) {
    pool[in] = nextp;
    in = (in+1) % n;
    count++;
}
```


Näide: puhver kriitiliste regioonidega (2)

- Tarbija:

```
region buffer when (count > 0) {
    nextc = pool[out];
    out = (out+1) % n;
  count--;
}
```

- Realisatsioon: mutex + semaforid tingimuse ootamiseks ja B taasväärtustamiseks

Monitorid

- Kah kõrgema taseme sünkroniseermiskonstruktsioon
- Laseb jagada operatsioone mingi andmetüübiga erinevate protsesside vahel

```
monitor monitor-name
{
  shared variable declarations
  procedure body P1 (...) { . . . }
  procedure body P2 (...) { . . . }
  procedure body Pn (...) { . . . }
  {
    initialization code
  }
}
```

Tingimusmuutujad

- Et protsess saaks oodata monitoris, on vaja kasutada tingimusmuutujaid:
  `condition x,y;`
- Tingimusmuutuja ainsateks operatsioonideks on `wait()` ja `signal()`:
  - `x.wait()` tähendab, et protsess blokeerub, kuni mõni teine protsess kutsub välja `x.signal()`
  - `x.signal()` tähendab, et äratatakse täpselt üks x taga ootel olev protsess. Kui selliseid pole, siis ei tehta midagi (võrdle semaforiga!)
- Monitoris tekib ootel protsesside järjekord. Järjekorras saab kasutada prioriteete — `wait(c)`

Näide: Solaris (1)

- Realiseerib mitmeid erinevaid tüüpe lukke, et toetada paljusid protsesse, paljusid lõimi (ka reaalajalõimi) ja paljusid protsessoreid.
- Kasutab adaptiivseid mutexeid (*busy wait* mitme protsessori korral, blokeerumine ühe protsessori korral) lühikeste koodilõikude üksteise eest kaitsmiseks
- Kasutab tingimusmuutujaid ja lugemis-kirjutamislukke pikemate koodilõikude jaoks

Näide: Solaris (2)

- Kasutab lukkude (adaptiivse mutexi ja rwlock'i) juures lõimede nimekirju (*turnstiles*), et meeles pidada luku järel ootavaid lõimi (+prior. pärimine)
- *Turnstile*'id pole seotud iga objektiga, vaid ainult nendega, kus on ootajaid
- Igal lõimel on oma *turnstile*. Kui ta on luku juures esimene blokeeruja, annab ta oma *turnstile*'i lukule ja kui vabaneb, saab tagasi. Kui keegi teine tema vabanemisel järjekorda jääks, siis antakse talle asemele kasutamata vaba *turnstile*.

Näide: Windows 2000 (1)

- Kasutab katkestuste maske üheprotsessrisüsteemidel ja spin-lukke (ingl. *spinlock*) mitmeprotsessorisüsteemidel
- Luku hoidjat välja ei tõrjuta
- Dispetšerobjektid — tuumaväliste lõimede sünkroniseermiseks (käituvad nagu mutexid või semaforid)
- Dispetšerobjektid võimaldavad kasutada ka sündmusi (*event*) lõimedele soovitud tingimustest teatamiseks (sarnane tingimusmuutujatele)

Näide: Windows 2000 (2)

- Dispetšerobjektid on kas signaliseeritud (vabad) või signaliseerimata (kasutamisel lõim blokeeruks) olekus
- Signaliseerimata olekus dispetšerobjekti taha blokeeruv lõim läheb objekti ootejärjekorda
- Objekti oleku muutusel signaliseerituks äratatakse järjekorrast üks (näiteks mutexil) või mitu (näiteks sündmusel) lõime