IN1805 I – Operating System Concepten

Hoofdstuk 6: Process synchronization

Proces Synchronisatie
Producer-consumer probleem revisited

shared

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

producer

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

consumer

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

De kritieke Sectie (1)

Definitie: Een kritieke sectie is een gebied in de code waarvoor geldt, dat als een proces zich in dat gebied bevindt, andere processen zich daar niet mogen bevinden, bovendien mogen andere processen zich niet in sommige andere kritieke secties bevinden.

```
        ...
entry section                (ingang van de sectie)
        kritieke sectie
exit section                 (uitgang van de sectie)
        ...
```

De kritieke Sectie (2)

Eisen aan de oplossing:

1. Wederzijdse uitsluiting:
   - Er mag hoogstens 1 proces tegelijk in de kritieke sectie
2. Progressie
   - Een proces buiten de kritieke sectie kan niet verhinderen dat een ander proces de kritieke sectie binnengaat
3. Wachttijd niet onbeperkt
   - Als een proces de kritieke sectie in wil, is er een grens aan het aantal processen dat er voor hem in mag.

De kritieke Sectie (3)

Algemene vorm voor proces P_i :

```
do {
```

entry Kritieke Sectie

Kritieke Sectie

exit Kritieke Sectie

Rest van de code

```
} while(1);
```

Kritieke Sectie (4)

Software oplossingen voor het Kritieke Sectie probleem verschillen in:

- gebruikte shared variabelen
- codering van ***entry*** en ***exit***

Kritieke Sectie
Oplossing voor 2 processen (1)

Processen P_0 en P_1, of P_i en P_j met j = 1 - i

'Oplossing' 1 (Code voor P_i)

```
shared  int turn=i;

entry   while (turn != i)
              /* do nothing */;
exit    turn = j;
```

- Wederzijdse uitsluiting gegarandeerd,
- Progressie niet gegarandeerd (toegang alternerend)

Kritieke Sectie
Oplossing voor 2 processen (2)

'Oplossing' 2 (code voor P_i)

shared `boolean` flag[2];

```
   /*    initieel flag[i]=false voor i=0..1   */
```

entry `flag[i] = true;`
`while (flag[j]);`

exit `flag[i] = false;`

- Wederzijdse uitsluiting gegarandeerd
- Progressie niet. (probleem als flag[0] en flag[1] beide true)

Kritieke Sectie
Oplossingen voor 2 processen (3)

Oplossing 3 (code voor P_i)

```
shared boolean flag[2];
       int turn = i;
   /*   initieel flag[i]= false voor i=0..1   */

entry  flag[i] = true;
       turn = j;
       while (flag[j] && turn == j);

exit   flag[i] = false;
```

- voldoet aan uitsluiting, progressie en beperkte wachttijd

Wederzijdse Uitsluiting(1)

Oplossingen:

1. Software oplossingen
   - Complex, niet vaak gebruikt
2. Hardware oplossingen
   - Disable interrupts
     - moeilijk voor multiprocessors
   - **Test-and-Set** instructie of **Swap** instructie (locking)
     - werken ook voor multiprocessors

Wederzijdse Uitsluiting (2)
Test-and-SET instructie

Test-and-Set is een machine instructie, dus *Atomair* !

Beschrijving:

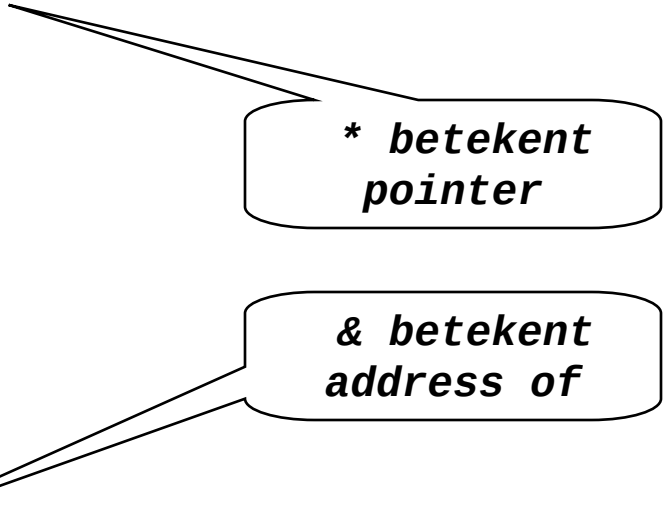
```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = true;
   return rv;
}
```

Gebruik :

```
shared  boolean lock=false;

entry   while (TestAndSet(&lock));

exit    lock = false;
```



Wederzijdse uitsluiting (3)
Swap instructie

Swap is een machine instructie, dus *Atomair* !

Beschrijving

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

Gebruik

```
shared  boolean lock = false;
entry   key = true;
        while (key == true) Swap(&lock, &key);

exit    lock = false;
```

Wederzijdse uitsluiting (4)

```
shared boolean waiting[N];          /*   initieel waiting[i]= false  */
       boolean lock = false;

entry  waiting[i] = true;
       key = true;
       while (waiting[i] && key==true) Swap(&lock, &key);
       waiting[i] = false;

exit   j = (i+1) % N;
       while (j != i && !waiting[j])
             j = (j+1) % N;
       if (j == i)
             lock = false;
       else
             waiting[j] = false;
```

PAUZE

Semaforen (1)

Voorlopige definitie:

Een semafoor is een integer variabele met 2 (atomaire) operaties:

- Wait (ook aangeduid als P, of als down)
- Signal (ook aangeduid als V, of als up)

```
Wait(S) {
    while  (S <= 0) ;                    /* busy waiting */
    S := S  - 1;
}
Signal (S) {
    S := S + 1;
}
```

Semaforen (2)

- Gebruik ten behoeve van Wederzijdse Uitsluiting:

  Semafoor *mutex* initieel 1

entry `Wait(mutex)`

exit `Signal(mutex)`

- Gebruik ten behoeve van synchronisatie:

  Stel B mag pas als A klaar is,

  Semafoor *synch* initieel 0

  A eindigt met: `Signal(synch)`

  B begint met `Wait(synch)`

Semaforen (3)

- Volgens voorlopige definitie:

  Wait operatie bevat busy waiting
- Het is meestal beter busy waiting te vermijden
- Nieuwe definitie:

  Semafoor bestaat uit een (integer) waarde plus een queue

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

Semaforen (4)

Herdefinitie van Wait en Signal

```
void wait(semaphore S){
    S.value = S.value - 1;
    if (S.value < 0) {
          add process to S.L;
          block();
    }
}
void signal(semaphore S) {
    S.value = S.value + 1;
    if (S.value <= 0) {
          remove a process P from S.L;
          wakeup(P);
    }
}
```

Semaforen (5)

Complicaties:

- Een semafoor mag maar door 1 proces tegelijk veranderd worden.
  - D.w.z. Wait en Signal zelf zijn ook weer **kritieke secties**
  - bescherming nodig, b.v. m.b.v. Test-and-Set
  - Busy waiting hiermee niet volledig uitgeschakeld, maar beperkt in de tijd
- Deadlock mogelijk
  - door verschillende volgorde van Wait in verschillende processen
- Starvation mogelijk
  - Bij LIFO toekenning (i.p.v. FIFO) bij Signal

Semaforen (6), voorbeeld: producer-consumer met bounded buffer

shared Semafoor mutex initieel (1)
Semafoor empty initieel (n) /* aantal lege buffers */
Semafoor full initieel (0) /* aantal volle buffers */

producer

```
do {

    produce an item in nextp

    wait(empty);
    wait(mutex);

    add nextp to buffer

    signal(mutex);
    signal(full);
} while(1);
```

consumer

```
do {

    wait(full);
    wait(mutex);

     move an item to nextc

    signal(mutex);
    signal(empty);

     consume the item in nextc

} while(1);
```

Semaforen (7)
voorbeeld: Readers and Writers probleem

Het probleem:

- shared buffer
- Readers zijn processen die alleen mogen lezen
- Writers zijn processen die mogen lezen of schrijven
- meerdere lezers kunnen tegelijk lezen
- schrijvers hebben exclusieve toegang nodig

varianten:

- *Eerste RW-probleem:* Readers hoeven alleen te wachten als writer toegang heeft gekregen.
- *Tweede RW probleem:* Als een writer toegang wil, moet hij deze zo snel mogelijk krijgen

Bij beide varianten bestaat het risco van *Starvation*

Semaforen(8)
Oplossing voor het eerste RW-probleem

shared semaphore mutex initial(1);
semaphore wrt initial(1)*;*
int readcount = 0;

Writer

wait(wrt);

hier wordt geschreven

signal(wrt);

Semaforen(9)
Oplossing voor het eerste RW-probleem

Reader

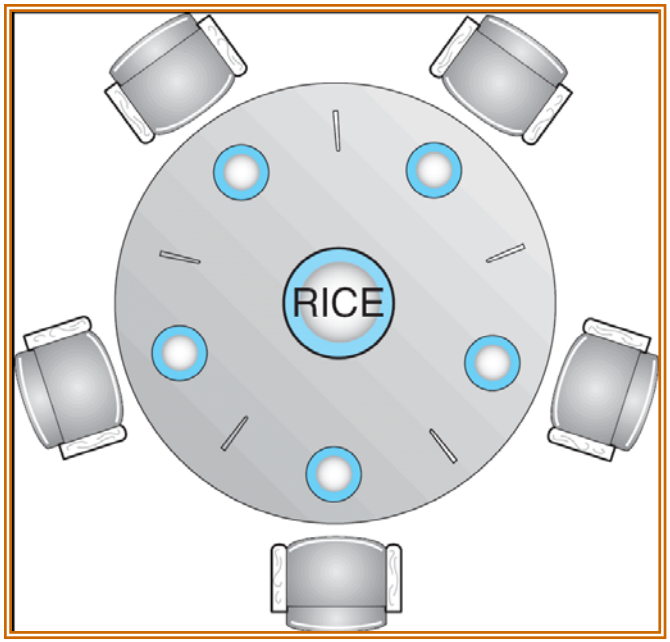
```
wait(mutex);
readcount = readcount + 1;
if (readcount == 1)
  wait(wrt); /* 1st reader wacht tot geen schrijver */
signal(mutex);
```

hier wordt gelezen

```
wait(mutex);
readcount := readcount - 1;
if (readcount == 0)
  signal(wrt);  /* laatste lezer meldt zijn vertrek */
signal(mutex);
```

Semaforen (10)
Dinerende filosofen

- Klassiek probleem
- eisen aan oplossing:
  - deadlock vrij
  - starvation vrij



Semaforen (11)
Dinerende filosofen

'Oplossing' 1: (code voor filosoof i)

```
shared semaphore chopstick[5];
    initieel chopstick[i]=1 voor i=0..4
do {
    wait(chopstick[i]);                    /* links */
    wait(chopstick[i+1 mod 5]);            /* rechts */
    ...     hier wordt gegeten     ...
    signal(chopstick[i]);
    signal(chopstick[i+1 mod 5]);
    ...   hier wordt gedacht    ...
} while(1);
```

- Deze oplossing is niet deadlock vrij;
- voorbeeld van deadlock vrij algoritme is de asymmetrische oplossing

Alternatieven voor semaforen

- Semaforen kunnen makkelijk verkeerd worden gebruikt
  - b.v. vergeten van Wait en/of Signal
  - b.v. verwisselen van Wait en Signal
- hogere-taal constructies om de kans op fouten te verminderen:
  - (conditional) critical region
  - monitors

Monitors

- Abstract data type met ingebouwde wederzijdse uitsluiting

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }
       …

  procedure Pn (…) {……}

   Initialization code (…) { … }
}
```

Condition variables

- `condition x, y;`

- Two operations on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended.
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`

Dinerende filosofen mbv monitors (1)

```
monitor DP {
   enum { THINKING, HUNGRY, EATING) state [5];
   condition synch [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) synch [i].wait;
   }

   void putdown (int i) {
          state[i] = THINKING;
                     // test left and right neighbors
          test((i + 4) % 5);
          test((i + 1) % 5);
   }
```

Dinerende filosofen mbv monitors (2)

```
  void test (int i) {
         if ( (state[(i + 4) % 5] != EATING) &&
              (state[i] == HUNGRY) &&
              (state[(i + 1) % 5] != EATING) ) {
             state[i] = EATING ;
             synch[i].signal () ;
         }
   }

   initialization_code() {
         for (int i = 0; i < 5; i++)
             state[i] = THINKING;
  }
}
```