Tentamen A2 (deel a) en PC : 15-04-2004

Vraag 1 (1 punt)

Wat is de uitvoer van het volgende C programma?

```
int i = 0;
int main() {
  i++; fork(); i++; fork(); i++; printf("i is %i \n", i);
}
```

Vraag 2 (1 punt)

Het schedulingsalgoritme van Windows 2000 biedt (ondermeer) de keuze tussen een kort en een lang quantum. Op general-purpose computers (i.e. in Windows 2000 Professional) wordt standaard gekozen voor een kort quantum, op servers (i.e. in Windows 2000 Server) voor een lang quantum. Waarom zou dit zo zijn? NB. beschouw beide keuzes.

Vraag 3 (2 punten)

Een single-threaded proces duurt 10 seconden. Hiervan zit het 2 seconden in de running toestand, 2 seconden in de ready toestand, en 6 seconden in de blocked toestand (vanwege disk I/O). Om het proces sneller laten lopen, stelt iemand voor het proces multi-threaded te maken.

(a) Zou dit iets uitmaken in het geval de computer 1 CPU heeft, en, zoja, wat is de maximaal mogelijke tijdwinst die we zouden kunnen halen? (dwz. in het allergunstigste geval.) Maakt het hierbij uit of we kernel- of user-level threads hebben?

(b) Zou dit iets uitmaken in het geval de computer twee CPU's heeft, en, zoja, wat is de maximaal mogelijke tijdwinst die we zouden kunnen halen? Maakt het hierbij uit of we kernel- of user-level threads hebben?

(c) Iemand anders komt met het voorstel om het proces op te splitsen in meerdere processen ipv. meerdere threads. Onder welke omstandigheden is dit voorstel beter dan wel slechter?

Motiveer je antwoorden.

Vraag 4 (1 punt)

We beschouwen wat varianten van round-robin scheduling:

(a) Bij variant (a) wordt het quantum van een proces verhoogd als het z'n quantum helemaal verbruikt. Lijkt je dit zinnig? Wat zijn mogelijke voor- en nadelen? Kan er starvation optreden?

(b) Bij variant (b) wordt het quantum van een proces verlaagd als het z'n quantum niet helemaal verbruikt. Lijkt je dit zinnig? Wat zijn mogelijke voor- en nadelen? Kan er starvation optreden?

Vraag 5 (1 punt)

We gebruiken Dekker's algoritme om mutual exclusion tussen twee processen T_1 en T_2 te garanderen. Stel T_1 zit in de kritieke sectie en T_2 wil de kritieke sectie binnengaan (maar kan dat dus niet). In welke van de toestanden – *Ready*, *Running*, of *Blocked* – kan T_1 zitten? En T_2 ? Motiveer je antwoorden.

Vraag 6 (2 punten)

We hebben processen `P1`, `P2`, `P3` die de volgende pseudo-code uitvoeren:

```
P1:  while(true) { s1 ; cs1 }

P2:  while(true) { cs2 ; s2 }

P3:  while(true) { s3 ; cs3 ; s4 }
```

De `s`i en `cs`i hier zijn stukken code die op zich uit meerdere instructies kunnen bestaan.

`P1` produceert data voor `P2`, en `P2` voor `P3`, dus we willen niet dat `P2` 'voorloopt' op `P1`, of dat `P3` 'voorloopt' op `P2`. Om precies te zijn, als `P`$i+1$ de body van zijn while-lus voor de n-de keer gaat uitvoeren, moet `P`i de body van zijn while-lus tenminste n keer hebben uitgevoerd, voor $i = 1, 2$. Verder moeten we mutual exclusion tussen `cs1` en `cs2` garanderen, en tussen `cs2` en `cs3`. Maar, als dat lukt, willen we wel toestaan dat `P1` met `cs1` bezig is terwijl `P3` met `cs3` bezig is.

Voeg semaforen aan de code toe om de beschreven mutual exclusion en synchronisatie te garanderen. Geef van de gebruikte semaforen hun initiële waarden, en zeg of het gewone dan wel binaire semaforen zijn.

NB. ER STAAT NOG EEN VRAAG ACHTEROP!

Vraag 7 (2 punten)

De volgende (pseudo)code implementeert message passing tussen threads, gebruik makend van een standaard bounded buffer. Messages zijn gewoon integers, om 't simpel te houden. Om precies te zijn, de code implementeert, zoals gebruikelijk, een blocking receive and een non-blocking send. Ook al kan de send blokkeren – namelijk bij een volle buffer –, het is wat we een non-blocking send noemen.

```
int buffer[N]; // buffer voor messages
int head = 0;  // positie van eerste message
int tail = 0;  // positie van laaste message
Semaphore a[2];
BinarySemaphore b;

void send( int m ) {
  a[0].wait();
  b.wait();
    buffer[tail] = m; tail = (tail + 1) % N;
  b.signal();
  a[1].signal();
}

int receive(){
  int m;
  a[1].wait();
  b.wait();
    m = buffer[head]; head = (head + 1) % N;
  b.signal();
  a[0].signal();
  return m;
}
```

Initiëel is `a[0].count==N`, `a[1].count==0`, en `b.count==1`.

(a) Maakt het uit of we de operaties `a[0].wait()` en `b.wait()` omwisselen in `send`? Maakt het uit of we de operaties `b.signal()` en `a[0].signal()` omwisselen in `receive`? Motiveer je antwoorden.

(b) Iemand stelt voor om ook een non-blocking receive te implementeren, die nooit blokkeert, maar die gewoon -1 teruggeeft als er geen bericht klaar staat om te ontvangen, namelijk als volgt:

```
int tryReceive(){
  b.wait();
  if (a[1].count == 0) { b.signal();
                         return -1;
                       }
  else { a[1].wait();
         m = buffer[head]; head = ( head + 1 ) % N;
         b.signal();
         a[0].signal();
         return m;
       }
  }
```

Is dit een correcte implementatie voor een non-blocking receive? Motiveer je antwoord.

(c) Geef de pseudo-code voor een blocking send operatie `void blockingSend(int m)`. Een aanroep van deze methode mag dus pas terugkeren als het gezonden bericht ontvangen is. Het moet mogelijk zijn aanroepen van deze `blockingSend` en de non-blocking `send` door elkaar te doen. Introduceer zonodig extra semaforen, en geef dan de initiële waarde, en zeg of het gewone dan wel binaire semaforen zijn. Pas zonodig de bestaande code van `send` en `receive` aan.

Semaforen zijn geimplementeerd zoals in Stallings staat beschreven, dwz. gewone semaforen kunnen een negatieve count krijgen, binaire semaforen hebben altijd count 0 of 1, en een signal op een binaire semafoor die al op 1 staat heeft geen effect.