Uitwerking Concurrency, 20 januari 2009

Tijdsduur 3 uur. Gesloten boek tentamen.

Opgave 1 (20 %).
(a) Specificeer *wederzijdse uitsluiting* (mutual exclusion) voor een systeem van N processen.
(b) Definieer de begrippen *semafoor* en *binaire semafoor*.
(c) Definieer het begrip *mutex*. Wat is voor het gebruik het belangrijkste verschil tussen mutexen en semaforen?
(d) Implementeer de wederzijdse uitsluiting van onderdeel (a) met behulp van semaforen.
(e) Implementeer de wederzijdse uitsluiting van onderdeel (a) met behulp van mutexen.

Uitwerking. (a) Wederzijdse uitsluiting voor een systeem van N processen betekent, dat de processen elk een stuk code (traditioneel *CS* geheten) hebben waarvoor op de één of andere manier bewerkstelligd wordt dat er nooit meer dan één process tegelijk in *CS* is. Dit wordt doorgaans gemodelleerd in gemeenschappelijke code van de vorm

$$\textbf{do } \mathit{true} \quad \rightarrow \quad \mathit{NCS}\ ;\ \mathit{CS} \quad \textbf{od}\ .$$

met de eis dat altijd geldt

$$\#\{q \mid q \textbf{ in } \mathit{CS}\} \leq 1\ .$$

(b) Een semafoor is een integer variabele, zeg `s`, die afgezien van initialisatie alleen benaderd kan worden met de atomaire operaties

$$\begin{array}{lcl} P(\texttt{s}) & = & \langle\ \textbf{await } \texttt{s} > 0 \textbf{ then } \texttt{s --}\ \rangle\ , \\ V(\texttt{s}) & = & \langle\ \texttt{s ++}\ \rangle\ . \end{array}$$

De semafoor heet binair als $\texttt{s} \leq 1$ gehouden moet en kan worden.

(c) Een mutex werkt net als een binaire semafoor die op 1 geïnitialiseerd is. De operaties heten resp. `lock` en `unlock`. Het belangrijkste verschil is, dat `unlock` gedaan moet worden door het process dat ge**lock**t heeft.

(d)
```
sem s := 1 ;
do true  →
   NCS ;
   P(s) ;
   CS ;
   V(s) ;
od .
```

(e)
```
var mu : Mutex ;
do true  →
   NCS ;
   lock(mu) ;
   CS ;
   unlock(mu) ;
od .
```

Opgave 2 (40 %). Beschouw een begrensde stapel voor een systeem van concurrent processen met twee procedures

procedure *push*(x : *Item*) ;
procedure *pop*() : *Item* .

De aanroep *push*(x) plaatst x bovenop de stapel. De aanroep *pop*() haalt het bovenste element van de stapel en levert dit op. Als de stapel leeg is, moet de procedure *pop* wachten tot een ander process iets gepusht heeft. De stapel kan N items bevatten. Als de stapel N items bevat, moet *push* wachten tot een ander process iets gepopt heeft.
(a) Geef hiervan een arrayimplementatie met behulp van gedeelde variabelen, atomiciteitshaakjes en (enkelvoudige of samengestelde) **await** statements.
(b) Zet de implementatie van onderdeel (a) om in een implementatie met standaard Java-primitieven. Je mag hierbij *niet* gebruik maken van de Java classen Semaphore en Mutex.
(c) Zet de implementatie van onderdeel (a) om in een implementatie met gesplitste binaire semaforen.

Uitwerking.

(a)
```
var stack : Item[N]
var free : Int := 0

procedure push(x : Item)
    ⟨ await free < N then stack[free] := x ; free ++ ⟩

procedure pop() : Item
    var result : Item
    ⟨ await free > 0 then free --  ; result := stack[free] ⟩
    return result
```

```
(b) synchronized void push(Item x) throws InterruptedException {
        while (free == N) wait() ;
        stack[free] = x ; free ++ ;
(*)     if (free == 1) notifyAll() ;
    }
    synchronized Item pop() throws InterruptedException {
        while (free == 0) wait() ;
        free -- ;
(*)     if (free == N-1) notifyAll() ;
        return stack[free] ;
    }
```

De conditionele aanroepen van `notifyAll` hebben tengevolge, dat er alleen threads in de wachtrij zijn als `free` $= 0$ (poppers) of *free* $= N$ (pushers) geldt. Het weglaten van de condities mag, maar is minder goed voor de performance.

De regels (*) mogen niet vervangen worden door "`notify();`". Dit zou namelijk ongerechtvaardigde deadlock geven, zoals blijkt in het volgende scenario. Stel dat er $4N$ threads zijn, waarvan er $2N$ een `push` moeten doen en $2N$ een `pop`. Neem aan dat $2N$ poppers eerst gescheduled worden; deze gaan dan alle de wachtrij in. Daarna worden de $2N$ pushers gescheduled. De eerste N pushers pushen en termineren; de stapel wordt hierdoor vol; N poppers worden uit de wachtrij gelaten door `notify`. De volgende N pushers gaan de wachtrij in. De vrijgelaten N poppers poppen, maken daarmee de stapel leeg, en laten de overgebleven N poppers vrij. Omdat de stapel leeg is, gaan deze poppers de wachtrij weer in. De stapel blijft leeg, de wachtrij bevat N pushers en N poppers, en er is ongerechtvaardigde deadlock.

(c)

sem gs := 1 , full := 0 , empty := 0
var fucnt : *Int* := 0 , emcnt : *Int* := 0

procedure *push*(x : *Item*)
  P(gs) ; fucnt ++ ; *Wissel*
  P(full) ; fucnt -- ; stack[free] := x ; free ++ ; *Wissel*
end *push*

procedure *pop*() : *Item*
  var *result* : *Item*
  P(gs) ; emcnt ++ ; *Wissel*
  P(empty) ; emcnt -- ; free -- ; *result* := stack[free] ; *Wissel*
  return *result*
end *pop*

Wissel:
  if free $< N \ \wedge$ fucnt $> 0 \ \rightarrow \ V$(full)
  $[\!]$ free $> 0 \ \wedge$ emcnt $> 0 \ \rightarrow \ V$(empty)
  $[\!]$ **else** $\rightarrow \ V$(gs)
  fi

Opgave 3 (40 %). Gegeven zijn $N > 1$ processen, alle in een oneindige lus

do *true* $\rightarrow$ *TNS* ; intro ; *ACT* ; exit **od** ,

waarbij *TNS* en *ACT* gegeven commando's zijn die gegarandeerd eindigen.

We stellen nu de volgende synchronisatie-eisen. Elk process q heeft een Boole'se privévariabele $bb.q$. Er dient altijd *ten hoogste* één proces r te zijn waarvoor $bb.r$ geldt. Er dient *precies* één dergelijk proces te zijn, wanneer *ACT* wordt uitgevoerd door welk proces dan ook. Als q *TNS* uitvoert dient $bb.q = false$ te zijn. Dit wordt uitgedrukt in de invarianten

(J0) $bb.q \ \wedge \ bb.r \ \Rightarrow \ q = r$,
(J1) q **at** *ACT* $\Rightarrow \ (\exists\, r : bb.r)$,
(J2) q **at** *TNS* $\Rightarrow \ \neg\, bb.q$.

Gegeven is een geheel getal $K > 1$. Een proces r waarvoor $bb.r$ geldt, dient $bb.r$ waar te houden tot tenminste K processen *ACT* uitgevoerd hebben en moet daarna de gelegenheid krijgen *TNS* weer uit te voeren.

Om dit voor elkaar te krijgen declareren we de gedeelde variabelen:

var cc : *Bool* := *true*
var cnt : *Int* := 0
var n : *Int* := 0

De processen worden geïmplementeerd volgens

process *Member*(*self* := 0 **to** $N - 1$)
  var bb : *Bool* := *false*
  do *true* $\rightarrow$
10: *TNS*
11: $\langle$ **await** cc **then** ... $\rangle$
12: $\langle$ bb := (n = 0) ; n ++ $\rangle$
13: *ACT*
14: ...
15: **if** bb $\rightarrow$
16: **await** n $\geq K$

```
17:                 ...
18:                 await  ?
19:                 ⟨ bb := false ; n := 0 ⟩
20:                 ...
                fi
            od end Member .
```

We eisen hiervoor de invariant

(J3) $\quad \texttt{cnt} = \#\{q \mid q \textbf{ in } \{12, 13, 14\}\}$.

We stellen de voortgangseis:

(vtg) $\quad q \textbf{ in } \{12, \ldots, 20\} \quad o\!\!\rightarrow \quad pc.q = 10$.

(a) Vul commando's in voor de puntjes in 11, 14, 17 en 20, en een conditie voor het vraagteken in 18, zodanig dat aan deze eisen voldaan wordt en dat er verder niet onnodig gewacht wordt.
(b) Toon aan dat je oplossing voldoet aan de invarianten (J0), (J1), (J2) en (J3). Je mag hiertoe andere invarianten invoeren en bewijzen.
(c) Laat zien dat je oplossing voldoet aan de voortgangseis (vtg).

Uitwerking. (a) We verhogen `cnt` in 11 en verlagen het in 14 om (J3) te krijgen. We zetten `cc` := *false* in 17 om te zorgen dat `cnt` terugloopt naar 0. We laten het proces r met $bb.r$ in 18 wachten tot `cnt` $= 0$. We zetten `cc` := *true* in 20 om de processen opnieuw toe te laten.

```
        process Member(self := 0 to N − 1)
            var bb : Bool := false
            do true  →
10:             TNS
11:             ⟨ await  cc  then  cnt ++ ⟩
12:             ⟨ bb := (n = 0) ; n ++ ⟩
13:             ACT
14:             ⟨ cnt -- ⟩
15:             if bb  →
16:                 await n ≥ K
17:                 cc := false
18:                 await  cnt = 0
19:                 ⟨ bb := false ; n := 0 ⟩
20:                 cc := true
                fi
            od end Member .
```

(b) (J3) volgt uit het feit dat `cnt` juist dan wordt opgehoogd als een proces 11 uitvoert, en juist dan verlaagd wordt als een proces 14 uitvoert. Om (J0) te bewijzen postuleren we de invarianten

(J4) $\quad bb.q \;\Rightarrow\; \texttt{n} \geq 1$,
(J5) $\quad q \textbf{ in } \{16, 17, 18, 19\} \;\Rightarrow\; bb.q$.

Invariantie van (J4) volgt uit (J0), (J5) en de code bij 12 en 19. Invariantie van (J5) volgt uit de code. Uit (J4) volgt dat een tweede proces nooit de gelegenheid krijgt om $bb.r$ te zetten voor q "vertrokken" is, oftewel (J0).

Invariant (J1) volgt uit (J4) en

(J6) $\quad \texttt{n} \geq 1 \;\Rightarrow\; (\exists\, r : bb.r)$,
(J7) $\quad pc.q = 13 \;\Rightarrow\; \texttt{n} \geq 1$.

Invariantie van (J6) volgt direct uit de code van 12 en 19.

De invariantie van (J7) is lastig. We hebben hiervoor (J2) en (J8) nodig:

(J8) $\quad q$ **in** $\{19, 20\} \;\Rightarrow\; \texttt{cnt} = 0$.

Voor de invariantie van (J8) hebben we nodig:

(J9) $\quad q$ **in** $\{18, 19, 20\} \quad \Rightarrow \quad \neg\texttt{cc}$.

Invariantie van (J9) volgt uit de invariant

(J10) $\quad \#\{r \mid r$ **in** $\{16, \ldots, 20\}\} \leq 1$.

Invariantie van (J10) volgt uit (J0), (J4), (J5) en

(J11) $\quad q$ **at** $20 \quad \Rightarrow \quad \texttt{n} = 0$.

Invariantie van (J11) volgt uit (J8) en (J3).

De invariant (J2) tenslotte volgt uit de sterkere maar triviale invariant

(J12) $\quad bb.q \quad \Rightarrow \quad q$ **in** $\{13, \ldots, 19\}$.

(c) Merk allereerst op dat in 1 ronde geldt q **at** $12\, o\!\!\rightarrow\, q$ **at** 13. In ten hoogste 3 ronden geldt

(vtg0) $\quad q$ **in** $\{13, 14, 15\} \;\wedge\; \neg\, bb.q \quad o\!\!\rightarrow \quad q$ **at** 10 ,

omdat q dan niet hoeft te wachten.

Als $bb.q$ in 13 geldt, loopt q door naar 16 en gaat daar wachten tot $\texttt{n} \geq K$ waar wordt. Dit gebeurt echter ooit en het blijft dan geldig, omdat n niet wordt gereset wegens (J10) en er steeds nieuwe processen bij 11 binnen komen, wegens de invariant

(J13) $\quad q$ **at** $16 \quad \Rightarrow \quad \texttt{cc}$.

Deze invariant volgt uit (J10) en de eenvoudige invariant

(J14) $\quad \neg\texttt{cc} \quad \Rightarrow \quad (\exists\, r : r$ **in** $\{18, 19, 20\})$.

Na uitvoering van 16 komt q via 17 bij 18, waar hij gaat wachten op $\texttt{cnt} = 0$. Dit gebeurt echter ooit wegens (J3) en (J9). Na uitvoering van 18 gaat q via 19 en 20 naar 10.

Er werd van jullie verwacht dat je een aantal van de sleutelinvarianten in je bewijs zou noemen.

Er geldt overigens niet: q **at** $11\, o\!\!\rightarrow\, q$ **at** 12. Als q at 11 is, wordt cc wel een keer waar, maar het kan weer onwaar zijn voordat q gescheduled wordt.