Példafeladatok

Tartalomjegyzék

Példafeladatok

1. Filozófusok

A filozófusok egy kerek asztal körül ülnek, közöttük egy-egy villa fekszik az asztalon. Az asztal közepén egy nagy tányéron van az étel, amelyből csak két villa felhasználásával tudnak enni. Mindenki csak a jobbján, illetve a balján lévő villákat használhatja. Minden filozófus a következő tevékenységeket ciklikusan végzi:

- gondolkodik,
- eszik.

A filozófus a villák helyzetével meghatározott állapotot azzal változtatja meg, hogy a balján *és* a jobbján levő villákat –*egyszerre*!– megragadja, ill. visszahelyezi.

A megoldás:

L1. Problémadefiníció:

Azonosítsa a **Philos(1..M)** a filozófusok „működési folyamatait". Erőforrás, amin meg kell osztozniuk: a villa, pontosabban a szomszédos filozófus(-folyamat)ok a köztük lévő villán osztoznak. Így kritikus szakaszban akkor tartozkodik egy folyamat, amikor két villát használ, azaz eszik. Az egyes folyamatokhoz –értelemszerűen– rendelhetünk egy-egy bináris szemafort, az alábbi értelmezéssel:

Legyen **eszik(1..M)** 1, ha az adott filozófus eszik, máskülönben 0.

A feladat feltételei szerint állandóan teljesülnie kell (azaz invariánsan igaz), hogy ha egy filozófus eszik, akkor ugyanezt nem teheti egyik szomszédja sem, másrészt viszont, ha ő nem eszik, akkor bármelyik szomszédja ehet (legalábbis ő miatta). Formálisan ugyanez:

$$\mathbf{Ph:}\ \forall i\in[1..M]:\ (eszik(i)=1 \wedge eszik(«i)+eszik(»i)=0) \vee (eszik(i)=0 \wedge 0\le eszik(«i)+eszik(»i)\le 2)$$ [1]

ahol a « és a » operátorok az *előző* és *következő index* kiválasztását végzik ciklikusan, azaz

»i = j, ahol j ≡ i+1 mod M, illetve

«i = j, ahol j ≡ i-1 mod M

L2. Megoldásvázlat

```
Változó     eszik: Tömb(1..M: Egész) (1..M: 0)      {a Ph triviálisan igaz, hisz enni nem kötelező}
Folyamat  Philosz(i: 1..M):
        Ciklus
            Gondolkodik
            {eszik(i)=0}   <eszik(i):=1>   {eszik(i)=1}
            Eszeget
            {eszik(i)=1}   <eszik(i):=0>   {eszik(i)=0}
        Ciklus vége
Folyamat vége.
```

[1] M≥4 esetén igaz.

L3. Az invariancia (további) biztosítása:

Vizsgálnunk kell, az alábbi atomi tevékenységek milyen feltétellel biztosítják a *Ph* szinkronizációs invariánst:

<eszik(i):=1> , <eszik(i):=0>

Lf(<eszik(i):=1>, Ph) ≡ (1)

Lf(<eszik(i):=1>, (2)
[«i. filozófus:] ((eszik(«i)=1 ∧ eszik(««i)+eszik(i)=0) ∨ (eszik(«i)=0 ∧ 0≤eszik(««i)+eszik(i)≤2)) ∧
[i. filozófus:] ((eszik(i)=1 ∧ eszik(«i)+eszik(»i)=0) ∨ (eszik(i)=0 ∧ 0≤eszik(«i)+eszik(»i)≤2)) ∧
[»i. filozófus:] ((eszik(»i)=1 ∧ eszik(i)+eszik(»»i)=0) ∨ (eszik(»i)=0 ∧ 0≤eszik(i)+eszik(»»i)≤2))
) ≡

≡[«i. filozófus:] ((eszik(«i)=1 ∧ eszik(««i)+1=0) ∨ (eszik(«i)=0 ∧ 0≤eszik(««i)+1≤2)) ∧ (3)
[i. filozófus:] ((1=1 ∧ eszik(«i)+eszik(»i)=0) ∨ (1=0 ∧ 0≤eszik(«i)+eszik(»i)≤2)) ∧
[»i. filozófus:] ((eszik(»i)=1 ∧ 1+eszik(»»i)=0) ∨ (eszik(»i)=0 ∧ 0≤1+eszik(»»i)≤2) ≡

≡ [«i. filozófus:] ((eszik(«i)=1 ∧ eszik(««i)=-1) ∨ (eszik(«i)=0 ∧ -1≤eszik(««i)≤1)) ∧ (4)
[i. filozófus:] ((1=1 ∧ eszik(«i)+eszik(»i)=0) ∨ (1=0 ∧ 0≤eszik(«i)+eszik(»i)≤2)) ∧
[»i. filozófus:] ((eszik(»i)=1 ∧ eszik(»»i)=-1) ∨ (eszik(»i)=0 ∧ -1≤eszik(»»i)≤1)) ≡

≡ [«i.filozófus:] (eszik(«i)=0) ∧ (5)
[i.filozófus:] (eszik(«i)+eszik(»i)=0) ∧
[»i.filozófus:] (eszik(»i)=0 ≡

≡ (eszik(«i)+eszik(»i)=0)

(1) ekvivalencia amiatt teljesül, mivel az i. filozófusnak közvetlen hatása csak két szomszédjára van, így elegendő a *Ph*-nak «i-re, i-re és »i-re vonatkozó részét vizsgálni. A (2) M1a) miatt igaz; (3) azonos átalakítást és az azonosan igaz részfeltételek elhagyását jelenti. (4)-nél figyelembe vettük az eszik(j) értékkészletére vonatkozó ismereteinket, továbbá elhagytuk az elhagyható tagjait a kifejezésnek. Így kapjuk, hogy

<várj amíg eszik(«i)+eszik(»i)≠0 majd eszik(i):=1>

Másrészt

Lf(<eszik(i):=0>, Ph) ≡ (1)

Lf(<eszik(i):=0>, (2)
[«i. filozófus:] ((eszik(«i)=1 ∧ eszik(««i)+eszik(i)=0) ∨ (eszik(«i)=0 ∧ 0≤eszik(««i)+eszik(i)≤2)) ∧
[i. filozófus:] ((eszik(i)=1 ∧ eszik(«i)+eszik(»i)=0) ∨ (eszik(i)=0 ∧ 0≤eszik(«i)+eszik(»i)≤2)) ∧
[»i. filozófus:] ((eszik(»i)=1 ∧ eszik(i)+eszik(»»i)=0) ∨ (eszik(»i)=0 Ù 0≤eszik(i)+eszik(»»i)≤2))
) ≡

≡ [«i. filozófus:] ((eszik(«i)=1 ∧ eszik(««i)+0=0) ∨ (eszik(«i)=0 ∧ 0≤eszik(««i)+0≤2)) ∧ (3)
[i. filozófus:] ((0=1 ∧ eszik(«i)+eszik(»i)=0) ∨ (0=0 ∧ 0≤eszik(«i)+eszik(»i)≤2)) ∧
[»i. filozófus:] ((eszik(»i)=1 ∧ 0+eszik(»»i)=0) ∨ (eszik(»i)=0 ∧ 0≤0+eszik(»»i)≤2)) ≡

≡ [«i. filozófus:] ((eszik(«i)=1 ∧ eszik(««i)=0) ∨ (eszik(«i)=0 ∧ 0≤eszik(««i)≤2)) ∧ (4)
[i. filozófus:] ((0≤eszik(«i)+eszik(»i)≤2)) ∧
[»i. filozófus:] ((eszik(»i)=1 ∧ eszik(»»i)=0) ∨ (eszik(»i)=0 ∧ 0≤eszik(»»i)≤2)) ≡

≡ [«i. filozófus:] ((eszik(«i)=1 ∨ eszik(««i)=0) ∨ (eszik(«i)=0)) ∧ (5)
[i. filozófus:] ∧
[»i. filozófus:] ((eszik(»i)=1 ∨ eszik(»»i)=0) ∨ (eszik(»i)=0)) ≡

≡ ¬(eszik(«i)=1 ∧ eszik(»i)=1) ≡ (6)

≡ Igaz

Most is a fentihez hasonló lépéseket végeztük el a (4)-ig. (5)-nél az $((A\wedge\neg B)\vee\neg A)\wedge((A\wedge\neg B)\vee\neg A)) \equiv \neg(A\wedge B)$ azonos átalakítást végeztük; a (6) a *Ph* invarianciáját tudva teljesül. Így kapjuk, hogy

< eszik(i):=0 >

L4. Az atomi tevékenységek implementálása*:*

Alkalmazzuk a *stafétabot továbbadása* módszert! Bevezetjük a **b** (belépési), a **c(1..M)** (*B(i)= eszik(«i)+eszik(»i)=0* feltételre várakozáshoz tartozó) bináris szemaforokat, valamint a **Db(1..M)** (*B(i)* feltételre várakozáshoz tartozó) számlálókat.

Változó **eszik: Tömb(1..M: Egész) (1..M: 0)** {a Ph triviálisan igaz, hisz enni nem kötelező}
b: BinSzemafor(1)
c: Tömb(1..M: BinSzemafor) (1..M: 0) {b+Σc(i)∈{0,1}, ui. BinSzem hasítás}
Db: Tömb(1..M: Egész) (1..M: 0) {Db(i)≥0}

Folyamat Philosz(i: 1..M):
Ciklus
Gondolkodik
P(b)
Ha* eszik(«i)+eszik(»i)'0 *akkor Db(i):=Db(i)+1; V(b); P(c(i))
eszik(i):=1*;*
***SIGNAL*$_1$**
Eszeget
P(b)
eszik(i):=0
***SIGNAL*$_2$**
Ciklus vége
Folyamat vége.

SIGNAL*$_i$*: ***Elágazás***
eszik(M)+eszik(2)=0 és Db(1)>0 esetén Db(1):=Db(1)-1; V(c(1))
eszik(1)+eszik(3)=0 és Db(2)>0 esetén Db(2):=Db(2)-1; V(c(2))
...
eszik(«i)+eszik(»i)=0 és Db(i)>0 esetén Db(i):=Db(i)-1; V(c(i))
...
eszik(M-1)+eszik(1)=0 és Db(M)>0 esetén Db(M):=Db(M)-1; V(c(M))
egyéb esetben V(b)
Elágazás vége

Megjegyzések:

1. Az 'egyéb'-feltétel gyanánt megfelel a $Db(1)=0 \wedge ... \wedge Db(M)=0$,
hiszen a $B(i)=(eszik(«i)+eszik(»i)=0)$ és $D(i)=(Db(i)>0)$ jelölésekkel az 'egyéb' nem más, mint $\neg(B(1)\wedge D(1))\wedge.. \wedge\neg(B(M)\wedge D(M))$, ami pedig $\neg(B(1)\vee...\vee B(M))\wedge\neg(D(1)\vee...\vee D(M))$, s az első tényező azonosan igaz volta miatt elhagyható, amiután már nyilvánvalóan a keresett formulánál vagyunk.

2. A *Db(i):=Db(i)+1* helyett *Db(i):=1* is elegendő, hisz csak az i. filozófus várhat a *B(i)* feltétel teljesülésére. Hasonló ok miatt a *Db(i):=Db(i)-1* helyett *Db(i):=0* is megfelelő.

3. Mivel a *Db(i):=1* a *P(c(i))*-vel, a *Db(i):=0* pedig a *V(c(i))*-vel együtt szerepel, ezért a programban a *c(i)=1-Db(i)* mindig igaz, így a *Db(i)>0* helyettesíthető a *c(i)=0* feltétellel, s a *Db*-k elhagyhatók a programból.[2]

[2] Ez is azt igazolja, hogy szerencsés lenne a W (van-e sorban álló?) szemafor-függvény bevezetése. Jelen esetben eleve csak a Db(i)>0 feltétel helyén szerepelne W(c(i)) alakban (azonos értelemben a c(i)=0 feltétellel).

A *holtpontmentesség*hez be kell látnunk: nem következhet be, hogy mindegyik filozófus várakozik a sorára, azaz $Db(1)>0 \wedge ... \wedge Db(M)>0$ (vagyis a $c(1)=0 \wedge ... \wedge c(M)=0$). Ami egyszerű következménye az algoritmus fenti szervezésének.

Ezen a ponton érdemes egy csöppet elidőzni, s meggondolni, hogy nem is csekélység az az elvárás, amit a szemaforokkal szemben támasztottunk. Ti. hogy: a kritikus szakaszba való többszörös belépés megelőzésére állítsa várakozási sorba a „többletfolyamatokat". Ez az eset –méghozzá különösen nehéz eset– lép föl akkor is, amikor *párhuzamosan* érkezik hozzá több folyamat. Milyen mechanizmusra van szükség ahhoz, hogy a valóban egyidőben érkező két, vagy több folyamat zavarmentesen vegye igénybe az oszthatatlan szemafor szolgáltatásait? Például tegyük föl, hogy a kritikus szakaszban éppen nem tartózkodik folyamat, amikor egyszerre két folyamat érkezik oda. Élesítsük tovább a kérdést! Hogy lehet az, hogy a kettő közül csak az egyiket engedi tovább, a másikat meg nem; az egyiknek foglaltat, a másiknak szabadot mutat, jól lehet ugyanakkor még semmilyen folyamat sincs a védett szakaszban? Nos, többféle mechanizmussal is megoldható ez a komoly kihívás. Egy ilyenről olvashatunk a függelékben.

2. „Balkezes" filozófusok

Az előző filozófus kerekasztal problémának tekintsük egy tipikus módosítását! Tegyük azt a kiegészítést, hogy „konfliktus esetben" a jobboldali[3] filozófussal szemben elsőbbsége legyen a tőle balra ülő szomszédjának. A „körbe várakozás" elkerülésére tegyük föl, hogy az 1-es számú filozófusnak mindkét szomszédjával szemben előnye van.

A megoldás:

L1. Problémadefiníció:

Most is a **Philos(1..M)** azonosítsa a filozófusok „működési folyamatait". Az előző feladathoz képest a módosítás az elsőbbség bevezetése; kérdés tehát az, hogy miként lehet ezt a „megbomlott" szimmetriát az invarianciában (s majd a folyamatok algoritmusaiban) figyelembe venni.

Ha megpróbáljuk követni az előző felaadat megoldásánál követett gondolatmenetet, előbb-utóbb rájövünk, hogy valami „újítást", nevezetesen a filozófusok egy „közbülső" állapotát kell bevezetnünk. Ez az állapot a filozófus azon *szándék*át fejezik ki, hogy enni óhajt. Lássuk be, hogy egy ilyen állapotra valóban szükség van.

Induljunk ki az előző megoldásalgoritmusból azt a módosítást elvégezve, hogy egy, az *elsőbbséget is* figyelembe vevő őrzőfeltételt (**B(i)**) illesztünk az ottani (**eszik(«i)+eszik(»i)≠0**) helyére.[4] Kövessük, mi történhet akkor, amikor két, szomszédos, „evésre jogosult" filozófus egyszerre akar enni. Mivel egyidőben érik el a **<várj amíg B(i) majd eszik(i):=1>** atomi műveletet felügyelő **P(b)** szemaforműveletet, egyikük beléphet az őrfeltétel-kiértékelő végrehajtási fázisba, a másik azonban a **b** szemafornál vár a belépésre. Hogy melyik, az nemdeterminisztikusan dől el. Ha történetesen a jobboldali ez a „szerencsés", akkor ő úgy érzékeli, hogy szabadon belefoghat az étkezésbe, s végrehajtja az **eszik(i):=1** műveletet. Amikor ezt követően „illedelmesen" átadja a *SIGNAL*-nál a stafétabotot a másiknak, az már a villát fölvett állapotban találja, így nem tehet másként, mint várja a villa felszabadulását. Így sérült meg a prioritás szabálya.

Tegyük föl már most, hogy minden filozófus a következő tevékenységeket végzi ciklikusan:

- gondolkodik,
- megéhezik (s bejelenti igényét a két mellette fekvő villára),
- eszik.

---

[3] Természetesen, nem politikai nézetre utal a jelző.

[4] A **B(i)**-től természetesen elvárjuk, kontrollálja, hogy *csak egy* filozófus juthat az adott villához.

A kritikus szakasz ez esetben meghosszabodott: elkezdődik már a „megéhezéssel”, s tart, míg a filozófus falatozik. Az egyes folyamatokhoz most rendeljünk egy-egy (nem bináris, hanem!) 3-értékű szemafort, az alábbi értelmezéssel:

Legyen **eszik(1..M)** 2, ha az adott filozófus eszik; 1, ha már megéhezett, de még nem jutott villához; 0 máskülönben.

A feladat feltételei szerint állandóan teljesülnie kell, hogy egy filozófus akkor ehet, ha jobboldali szomszédja nem eszik és a baloldali szomszédja még csak nem is éhezik, másrészt viszont, ha ő nem eszik, akkor bármelyik szomszédja ehet (legalábbis ő miatta). Az 1. és az M. filozófus esetében kicsit más a helyzet: az 1. mindaddig ehet, amíg szomszédai „legfeljebb” éheznek. Persze a gondolkodást és a megéhezést nem kell korlátozzuk a szinkronizációs invariánsban. Formálisan ugyanez:

$$\textbf{Ph: } \forall i\in[1..M]\textbf{: eszik(i)}\in\textbf{[0..2]} \wedge$$
$$\forall i\in[2..M-1]\textbf{: ((eszik(i)=2} \wedge \textbf{eszik(«i)=0} \wedge \textbf{eszik(»i)}\le\textbf{1) Ú eszik(i)<2)} \wedge$$
$$\textbf{((eszik(1)=2} \wedge \textbf{eszik(M)}\le\textbf{1} \wedge \textbf{eszik(2)}\le\textbf{1)} \vee \textbf{eszik(1)<2)} \wedge$$
$$\textbf{((eszik(M)=2} \wedge \textbf{eszik(M-1)<1} \wedge \textbf{eszik(1)<1)} \vee \textbf{eszik(M)<2)}$$

ahol a « és a » operátorok ugyanazok, mint az előző feladat megoldásánál volt.

L2. Megoldásvázlat

Változó **eszik: Tömb(1..M: Egész) (1..M: 0)** {a Ph triviálisan igaz, hisz enni nem kötelező}

Folyamat **Philosz(i: 1..M):**
Ciklus
Gondolkodik
{eszik(i)=0} **<eszik(i):=1>** **{eszik(i)=1}**
Megéhezett
{eszik(i)=1} **<eszik(i):=2>** **{eszik(i)=2}**
Eszeget
{eszik(i)=2} **<eszik(i):=0>** **{eszik(i)=0}**
Ciklus vége
Folyamat vége.

L3. Az invariancia (további) biztosítása:

Vizsgálnunk kell, az alábbi atomi tevékenységek milyen feltétellel biztosítják a *Ph* szinkronizációs invariánst:

<eszik(i):=2> , **<eszik(i):=1>** , **<eszik(i):=0>**

Mivel a *Ph* i függvényében némileg eltérő, ezért külön végezzük a levezetéseket az egyes esetekre.

$\forall i\in[2..M-1]$ eset:

Lf(<eszik(i):=2>, Ph) ≡

Lf(<eszik(i):=2>,
[«i. filozófus:] (eszik(«i)=2 ∧ eszik(««i)=0 ∧ eszik(i)≤1) ∨ (eszik(«i)<2) ∧
[i. filozófus:] (eszik(i)=2 ∧ eszik(«i)=0 ∧ eszik(»i)≤1) ∨ (eszik(i)<2) ∧
[»i. filozófus:] (eszik(»i)=2 ∧ eszik(i)=0 ∧ eszik(»»i)≤1) ∨ (eszik(»i)<2) ≡

≡ [«i. filozófus:] (eszik(«i)=2 ∧ eszik(««i)=0 ∧ 2≤1) ∨ (eszik(«i)<2) ∧
[i. filozófus:] (2=2 ∧ eszik(«i)=0 ∧ eszik(»i)≤1) ∨ (2<2) ∧
[»i. filozófus:] (eszik(»i)=2 ∧ 2=0 ∧ eszik(»»i)≤1) ∨ (eszik(»i)<2) ≡

≡ eszik(«i)=0 ∧ eszik(»i)≤1

Most a levezetés egyes lépéseit nem kommentáljuk. Így kapjuk, hogy

<várj amíg eszik(«i)≠0 vagy eszik(»i)=2 majd eszik(i):=2>

i=1 eset:

Lf(<eszik(1):=2>, Ph) ≡

Lf(<eszik(1):=2>,
[M. filozófus:] (eszik(M)=2 ∧ eszik(M-1)=0 ∧ eszik(1)≤1) ∨ (eszik(M)<2) ∧
[1. filozófus:] (eszik(1)=2 ∧ eszik(M)≤1 ∧ eszik(2)≤1) ∨ (eszik(1)<2) ∧
[2. filozófus:] (eszik(2)=2 ∧ eszik(1)=0 ∧ eszik(3)≤1) ∨ (eszik(2)<2) ≡

≡ [M. filozófus:] (eszik(M)=2 ∧ eszik(M-1)=0 ∧ 2≤1) ∨ (eszik(M)<2) ∧
[1. filozófus:] (2=2 ∧ eszik(M)≤1 ∧ eszik(2)≤1) ∨ (2<2) ∧
[2. filozófus:] (eszik(2)=2 ∧ 2=0 ∧ eszik(3)≤1) ∨ (eszik(2)<2) ≡

≡ eszik(M)≤1 ∧ eszik(2)≤1

A levezetés egyes lépéseit most sem kommentáljuk. Az eredmény:

<várj amíg eszik(M)=2 vagy eszik(2)=2 majd eszik(1):=2>

i=M eset:

Lf(<eszik(M):=2>, Ph) ≡

Lf(<eszik(i):=2>,
[M-1. filozófus:] (eszik(M-1)=2 ∧ eszik(M-2)=0 ∧ eszik(M)≤1) ∨ (eszik(M-1)<2) ∧
[M. filozófus:] (eszik(M)=2 ∧ eszik(M-1)<1 ∧ eszik(1)<1) ∨ (eszik(M)<2) ∧
[1. filozófus:] (eszik(1)=2 ∧ eszik(M)≤1 ∧ eszik(2)≤1) ∨ (eszik(1)<2) ≡

≡ [M-1. filozófus:] (eszik(M-1)=2 ∧ eszik(M-2)=0 ∧ 2≤1) ∨ (eszik(M-1)<2) ∧
[M. filozófus:] (2=2 ∧ eszik(M-1)<1 ∧ eszik(1)<1) ∨ (2<2) ∧
[1. filozófus:] (eszik(1)=2 ∧ 2≤1 ∧ eszik(2)≤1) ∨ (eszik(1)<2) ≡

≡ eszik(M-1)=0 ∧ eszik(1)=0

A levezetés egyes lépéseit most sem kommentáljuk. Az eredmény:

<várj amíg eszik(M-1)>0 vagy eszik(1)>0 majd eszik(M):=2>

A megéhezést jelentő atomi tevékenység őrfeltétele a következőképpen alakul ki:

∀i∈[2..M-1] eset:

Lf(<eszik(i):=1>, Ph) ≡

Lf(<eszik(i):=1>,
[«i. filozófus:] (eszik(«i)=2 ∧ eszik(««i)=0 ∧ eszik(i)≤1) ∨ (eszik(«i)<2) ∧
[i. filozófus:] (eszik(i)=2 ∧ eszik(«i)=0 ∧ eszik(»i)≤1) ∨ (eszik(i)<2) ∧
[»i. filozófus:] (eszik(»i)=2 ∧ eszik(i)=0 ∧ eszik(»»i)≤1) ∨ (eszik(»i)<2) ≡

≡ [«i. filozófus:] (eszik(«i)=2 ∧ eszik(««i)=0 ∧ 1≤1) ∨ (eszik(«i)<2) ∧
[i. filozófus:] (1=2 ∧ eszik(«i)=0 ∧ eszik(»i)≤1) ∨ (1<2) ∧
[»i. filozófus:] (eszik(»i)=2 ∧ 1=0 ∧ eszik(»»i)≤1) ∨ (eszik(»i)<2) ≡

≡ Igaz (1)

Csak az (1)-hez fűzünk rövid megjegyzést: ez a *Ph* invaranciája miatt teljesül. Következmény, hogy

< eszik(i):=1 >

A másik két eset levezetésén semmit sem változtat az egyes tényezőkben az = helyetti ≤, ill. <. A harmadik atomi utasítás őrfeltételének levezetése pontosan az előbbi lépesek szerint történhet.

Lf(<eszik(i):=0>,Ph) ≡ Igaz

Vagyis

< eszik(i):=0 >

L4. Az atomi tevékenységek implementálása:

Megint alkalmazzuk a *stafétabot továbbadása* módszert! Bevezetjük a **b** (belépési), a **c(1..M)** (*B(i)* a megfelelő feltételre várakozáshoz tartozó) bináris szemaforokat, valamint a **Db(1..M)** (*B(i)* feltételre várakozáshoz tartozó) számlálókat. Az előző megoldás megjegyzései itt is igazak úgy, hogy ennek ismeretében a végleges megoldás a következő, amiben már nem szerepelnek a Db-k:

```
Változó   eszik: Tömb(1..M: Egész) (1..M: 0)      {a Ph triviálisan igaz, hisz enni nem kötelező}
          b: BinSzemafor(1)
          c: Tömb(1..M: BinSzemafor) (1..M: 0)    {b+Σc(i)∈{0,1}, ui. BinSzem hasítás}

Folyamat Philosz(i: 2..M-1):
    Ciklus
        Gondolkodik
        P(b)
        eszik(i):=1
        SIGNAL_1
        Megéhezett
        P(b)
        Ha eszik(«i)≠0 vagy eszik(»i)=2 akkor V(b); P(c(i))
        eszik(i):=2;
        SIGNAL_2
        Eszeget
        P(b)
        eszik(i):=0
        SIGNAL_3
    Ciklus vége
Folyamat vége.

Folyamat Philosz1:
    Ciklus
        Gondolkodik
        P(b)
        eszik(1):=1
        SIGNAL_4
        Megéhezett
        P(b)
        Ha eszik(M)=2 vagy eszik(2)=2 akkor V(b); P(c(1))
        eszik(1):=2;
        SIGNAL_5
        Eszeget
        P(b)
        eszik(1):=0
        SIGNAL_6
    Ciklus vége
Folyamat vége.

Folyamat PhiloszM:
    Ciklus
        Gondolkodik
```

P(b)
eszik(M):=1
SIGNAL $_7$

Megéhezett

P(b)
Ha* eszik(M-1)>0 vagy eszik(1)>0 *akkor V(b); P(c(M))
eszik(M):=2*;*
SIGNAL $_8$

Eszeget

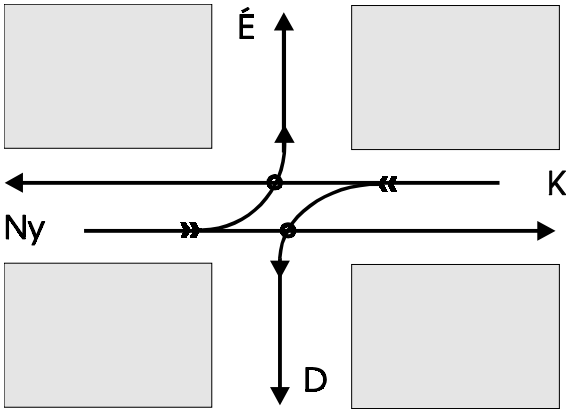
P(b)
eszik(M):=0
SIGNAL $_9$

Ciklus vége
Folyamat vége.

SIGNAL $_i$***: Elágazás***
eszik(M)≠2 és eszik(2)≠2 és c(1)=0 esetén V(c(1))
eszik(1)≠2 és eszik(3)≠2 és c(2)=0 esetén V(c(2))
...
eszik(«i)≠2 és eszik(»i)≠2 és c(i)=0 esetén V(c(i))
...
eszik(M-1)≠2 és eszik(1)≠2 és c(M)=0 esetén V(c(M))
egyéb esetben V(b)
Elágazás vége

3. Útkereszteződés jobbkéz szabály nélkül – egy „naív” megoldás

Kétirányú haladást lehetővé tevő útról az útkereszteződésben egy olyan egyirányú utca ágazik le jobbra is és balra is, amelybe az útról nagyívben lehetséges a behajtás. Nincs jobbkéz szabály, sőt sem rendőr, sem lámpa nincs a közelben.



Megoldás:

L1. Problémadefiníció:

Számunkra izgalmas az az eset, amikor valamelyik autó a kereszteződéshez ér. Egyéb mozgást most nem követünk. Így az egyes autók tevékenysége az alábbiakra korlátozódik:

- halad (a kereszteződés felé),
- megérkezik a kereszteződéshez,
- belép a kereszteződésbe (és a megfelelő irányba halad).

Irányonként egy sáv van. Konfliktus a két –a főúton– szembe jövő között alakulhat ki a kanyarodáskor, amikoris legalább az egyik keresztezi a másik útját. A négy helyzet közül kettő problematikus. A

tiltott esetek számbavételénél rövidítést alkalmazunk. Például egy keletről (K) jövő és nyugatnak (Ny) tartó autót röviden: K→Ny szimbólumokkal jelöljük. Tehát:

1. K→Ny & Ny→É (a 2. nagyívben keresztezi az 1. útját),
2. K→D & Ny→K (az 1. nagyívben keresztezi a 2. útját).

Megkülönböztetendő itt négyféle autófolyam(at), amelyeket haladási irányukkal jelölünk: **NyK** – a nyugatról keletfelé haladók, **NyÉ** – a nyugatról északfelé haladók, stb. A folyamatok a kereszteződésen osztoznak. A legegyszerűbb hozzáállás az, hogy feltesszük: a kereszteződést *egyidőben csak egy* autó használhatja. Ekkor ez az általános részbeli 3.1. szerinti módszerrel megoldható. A megoldás tehát:

```
Változó kerben: BinSzemafor(1)          {Invariáns: legfeljebb egy autó a kereszteződésben}

Folyamat  KNy(i: 1..DbKNy):
    halad(Ny{felé})
    P(kerben);  keresztez;  V(kerben)
    halad(Ny{felé})
Folyamat vége.

Folyamat  KD(i: 1..DbKD):
    halad(Ny{felé})
    P(kerben);  keresztez;  V(kerben)
    halad(D{felé})
Folyamat vége.

... a másik két folyamatfajta hasonló ...

Eljárás  halad(felé : Irány):
    ...
Eljárás vége.

Eljárás  keresztez:
    ...
Eljárás vége.
```

Ez a megoldás egy meglehetősen leegyszerűsített modellen alapszik, lépjünk ezen túl!

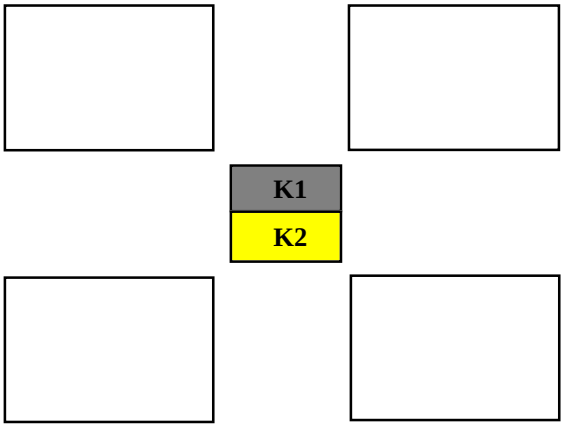
4. Útkereszteződés jobbkéz szabály nélkül – egy hatékonyabb megoldás

A fenti feladatot oldjuk meg úgy, hogy a valóságtól nagyon komolyan eltérő feltételezést, hogy *egyszerre csak* egy autó lehet a kereszteződésben, lazítjuk. Engedjük meg, hogy a két szemben levő sávon közlekedő autó –ha lehetséges– egyidőben osztozzanak a kereszteződésen.

Megoldás:

L1. Problémadefiníció:

A fent megfogalmazott „feltétel-lazítás” azt igényli tőlünk, hogy a kereszteződést „finomabb” szerkezetben bocsássuk az autók rendelkezésére. Figyelembevéve a feladat konkrét feltételeit, praktikusnak tűnik két részre bontani a főút két sávja mentén.



Ekkor egy **KNy** folyamat a kritikus szakaszba jutva kisajátítja a **K1**-t (de csak azt), egy **NyK** a **K2**-t, s bármely nagyívben kanyarodó (**KD**, **NyÉ**) mindkettőt. Legyen **fogl1**, **fogl2** a **K1**, ill. a **K2** keresztezési régióban levő autók száma. Elvárjuk annak teljesülését, hogy legfeljebb egy autó legyen akár a **K1**-ben, akár a **K2**-ben:

UK: 0≤fogl1≤1 ∧ 0≤fogl2≤1

L2. Megoldásvázlat:

Változó fogl1,fogl2: Egész(0) {UK}

Folyamat KNy(i: 1..DbKNy):
halad(Ny{felé})
<fogl1:=fogl1+1>; keresztez; <fogl1:=fogl1-1>
halad(Ny{felé})
Folyamat vége.

Folyamat NyK(i: 1..DbNyK):
halad(K{felé})
<fogl2:=fogl2+1>; keresztez; <fogl2:=fogl2-1>
halad(K{felé})
Folyamat vége.

Folyamat KD(i: 1..DbKD):
halad(Ny{felé})
<fogl1:=fogl1+1; fogl2:=fogl2+1>; keresztez; <fogl1:=fogl1-1; fogl2:=fogl2-1>
halad(D{felé})
Folyamat vége.

... a többi folyamatfajta ...

... a folytatás a korábbihoz hasonló ...

L3. Az invariancia (további) biztosítása:

Vizsgálnunk kell, az alábbi atomi tevékenységek milyen feltétellel biztosítják a *UK* szinkronizációs invariánst:

<fogl1:=fogl1+1>, <fogl2:=fogl2+1>, <fogl1:=fogl1+1; fogl2:=fogl2+1>
<fogl2:=fogl2-1>, <fogl1:=fogl1-1>, <fogl1:=fogl1-1; fogl2:=fogl2-1>

Lf(<fogl1:=fogl1+1>, UK) ≡

≡ Lf(<fogl1:=fogl1+1>, 0≤fogl1≤1 ∧ 0≤fogl2≤1) ≡

≡ 0≤fogl1+1≤1 ∧ 0≤fogl2≤1 ≡

≡ -1≤fogl1≤0 ∧ 0≤fogl2≤1 ≡ (1)

≡ fogl1=0

Magyarázatra csak (1) szorul: kihasználtuk ebben a lépésben, hogy *fogl1≥0* és a művelet előtt *UK* igaz volt; emiatt a második tényező elhagyható, az első meg egyszerűsíthető. Így kapjuk, hogy

<várj amíg fogl1≠0 majd fogl1:=fogl1+1>

hasonlóan kapható, hogy

<várj amíg fogl2≠0 majd fogl2:=fogl2+1>

illetve, hogy

<várj amíg fogl1≠0 vagy fogl2≠0 majd fogl1:=fogl1+1; fogl2:=fogl2+1>

Másrészt

Lf(<fogl1:=fogl1-1>, UK) ≡

≡ Lf(<fogl1:=fogl1-1>, 0≤fogl1≤1 ∧ 0≤fogl2≤1) ≡

≡ 0≤fogl1-1≤1 ∧ 0≤fogl2≤1 ≡

≡ 1≤fogl1≤2 ∧ 0≤fogl2≤1 ≡ (1)

≡ Igaz

Ismét csak (1)-t kell magyaráznunk: kihasználtuk ebben a lépésben, hogy *fogl1≥1* és a művelet előtt *UK* igaz volt; emiatt a második tényező elhagyható, az első meg egyszerűsíthető, sőt az alkalmazás szituációja miatt a *fogl1≤2* is nyilvánvalóan igaz. Így jön ki, hogy

< fogl1:=fogl1-1 >

< fogl2:=fogl2-1 >

< fogl1:=fogl1-1; fogl2:=fogl2-1 >

L4. Az atomi tevékenységek implementálása:

Ebben az esetben is az őrfeltételek keresztül-kasul hálózzák a folyamatokat, ami nem engedi, hogy valamilyen egyszerű, egy-két szemaforos megoldást adjunk. Alkalmaznunk kell a jólbevált *stafétabot továbbadása* módszert. Legyen a **b**, **d1**, **d2**, **d12** szemaforok rendre a belépéshez, a *(fogl1=0)*, a *(fogl2=0)*, ill. a *(fogl1=0 és fogl2=0)* feltételekhez, továbbá a **Db1**, **Db2**, **Db12** a megfelelő számlálók. Így a megoldás:

```
Változó     fogl1,fogl2: Egész(0)                                  {UK}
            b: BinSzemafor(1)
            d1,d2,d12: BinSzemafor(0)        {b+d1+d2+d12∈{0,1}, ui. BinSzemafor hasítás}
            Db1,Db2,Db12: Egész(0)

Folyamat  KNy(i: 1..DbKNy):
   halad(Ny{felé})

   P(b)     {UK}
   Ha fogl1≠0 akkor Db1:=Db1+1; V(b); P(d1)
   {UK ∧ fogl1=0}
   fogl1:=fogl1+1
   SIGNAL1

   keresztez

   P(b)     {UK}
   fogl1:=fogl1-1  {UK}
   SIGNAL2
```

```
    halad(Ny{felé})
Folyamat vége.
Folyamat  NyK(i: 1..DbNyK):
    halad(K{felé})
    P(b)      {UK}
    Ha fogl2≠0 akkor Db2:=Db2+1; V(b); P(d2)
    {UK ∧ fogl2=0}
    fogl2:=fogl2+1
    SIGNAL3
    keresztez
    P(b)      {UK}
    fogl2:=fogl2-1  {UK}
    SIGNAL4
    halad(K{felé})
Folyamat vége.
Folyamat  KD(i: 1..DbKD):
    halad(Ny{felé})
    P(b)      {UK}
    Ha fogl1≠0 vagy fogl2≠0 akkor Db12:=Db12+1; V(b); P(d12)
    {UK ∧ fogl1=0 ∧ fogl2=0}
    fogl1:=fogl1+1; fogl2:=fogl2+1
    SIGNAL5
    keresztez
    P(b)      {UK}
    fogl1:=fogl1-1; fogl2:=fogl2-1  {UK}
    SIGNAL6
    halad(D{felé})
Folyamat vége.
... a többi folyamatfajta ...
SIGNALi:  Elágazás
            fogl1=0 és Db1>0                  esetén  Db1:=Db1-1; V(d1)
            fogl2=0 és Db2>0                  esetén  Db2:=Db2-1; V(d2)
            fogl1=0 és fogl2=0 és Db12>0 esetén  Db12:=Db12-1; V(d12)
                  egyéb                       esetben V(b)
          Elágazás vége
... a folytatás a korábbihoz hasonló ...
```

Megjegyzések:

1. Hogy melyik autó kerül az útkereszteződésbe beléptetésre, akkor amikor többnek is módja lenne, az a *SIGNAL*-ban dől el. Nem determinisztikusan kerül kiválasztásra a több lehetséges eset közül egy. Tehát nincs kontrollálva semmifajta *jobbkéz szabállyal*.

2. Most nem foglalkozunk az egyes *SIGNAL*-programrészletek optimalizálásával.

3. Ebben a modellben nem akaszt meg a továbbhaladásban egy pl. egyenesen tovahaladót egy korábban érkező, de bekanyarodni nem tudó autó! Hiszen az egyes várakozási sorokban levők közül az fog először kilépni, amelyikre előbb kerül sor a *SIGNAL*-beli valamelyik ágban, nem számít az időbeliség.

   Gondoljuk végig egy példán e jelenséget! Ha nyugatról érkezik két autó, és az első észak felé, a második egyenesen kíván továbbmenni. Az elsőt megakadályozza egy keletről nyugat felé haladó autó a kanyarodásban. Annak ellenére, hogy a modell előzést nem ismer, mégis a második, egyenesen közlekedő beléphet az útkereszteződésbe. Ok: az akadályt képező **fogl1**-t és **fogl2**-t csak akkor képes

egy folyamat állítani, ha már bejutott a kritikus szakaszba, viszont amíg a várakozási sorban áll, addig nem. Így az egyik sorban állókat egy másik sorban álló kikerülheti, megelőzheti. A megoldás: az időbeliséget is adminisztráló explicit sor-struktúra.

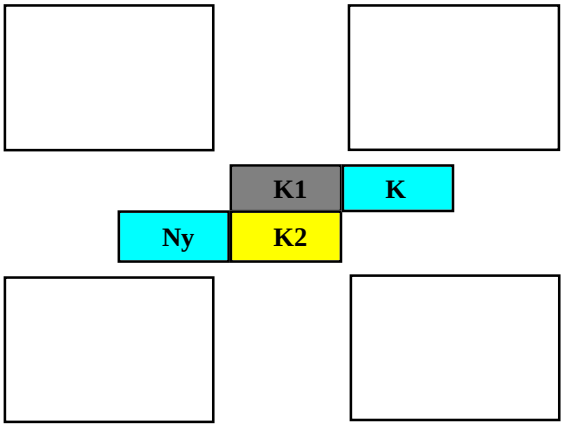
5. Útkereszteződés jobbkéz szabály nélkül – egy élethűbb megoldás

Az alapfeladatot abból a célból vizsgáljuk meg, hogy miként kerülhető ki az előző megoldás végén, a 3. megjegyzésben említett probléma.

Megoldás:

L1. Problémadefiníció:

A megoldás ötlete, hogy mindkét sávon a kereszteződés előtt (is) definiáljunk egy számlálót, amely az odaérkezőket –továbhaladási iránytól függetlenül!– adminisztrálja. Ha garantáljuk, hogy ott csak egy autó tartózkodhat egy időben, akkor már helyben is vagyunk.



Folytassuk a gondolatot a korábbi úton: bevezetjük a **K** és az **Ny** cellában levők számlálására a **foglK** és **foglNy** számlálókat. Elvárjuk annak teljesülését, hogy legfeljebb egy autó legyen akár **K1**-ben, **K2**-ben, akár **K**-ban, ill. **Ny**-ben:

$$\mathbf{UK:\ 0 \le fogl1 \le 1 \wedge 0 \le fogl2 \le 1 \wedge 0 \le foglK \le 1 \wedge 0 \le foglNy \le 1}$$

L2. Megoldásvázlat:

```
Változó fogl1,fogl2,foglK,foglNy: Egész(0)                    {UK}

Folyamat  KNy(i: 1..DbKNy):
    halad(Ny{felé})
    <foglK:=foglK+1>;  <fogl1:=fogl1+1>;  <foglK:=foglK-1>
    keresztez
    <fogl1:=fogl1-1>
    halad(Ny{felé})
Folyamat vége.

Folyamat  NyK(i: 1..DbNyK):
    halad(K{felé})
    <foglNy:=foglNy+1>;  <fogl2:=fogl2+1>;  <foglNy:=foglNy-1>
    keresztez
    <fogl2:=fogl2-1>
    halad(K{felé})
Folyamat vége.
```

Folyamat KD(i: 1..DbKD):
halad(Ny{felé})
<foglK:=foglK+1>; <fogl1:=fogl1+1; fogl2:=fogl2+1>; <foglK:=foglK-1>
keresztez
<fogl1:=fogl1-1; fogl2:=fogl2-1>
halad(D{felé})
Folyamat vége.
... a folytatás részben mechanikus kapható meg, részben változatlan az előzőhöz képest ...

Mint látható, először az autó belép abba a cellába, melybe minden adott irányból érkezőnek be kell lépnie, majd csak ezután foghat a kanyarodásba azzal, hogy a megfelelő adminisztrációt elvégzi. Ha ezen is túljutott, ez éppen azt jelenti, hogy elhagyhatja a „belépést" megelőző cellát, átengedve azt a következőnek.

L3. Az invariancia (további) biztosítása:

Vizsgáljuk az újonnan bevezetett négy atomi tevékenységet az *UK* invarianciája szempontjából:

<foglK:=foglK+1>, < foglNy:=foglNy+1>, <foglK:=foglK-1>, < foglNy:=foglNy-1>

Lf(<foglK:=foglK+1>, UK) ≡

≡ Lf(<foglK:=foglK+1>, 0≤fogl1≤1 ∧ 0≤fogl2≤1 ∧ 0≤foglK≤1 ∧ 0≤foglNy≤1) ≡

≡ 0≤fogl1≤1 ∧ 0≤fogl2≤1 ∧ 0≤foglK+1≤1 ∧ 0≤foglNy≤1 ≡

≡ 0≤fogl1≤1 ∧ 0≤fogl2≤1 ∧ -1≤foglK≤0 ∧ 0≤foglNy≤1 ≡ (1)

≡ foglK=0

Magyarázatra csak (1) szorul: kihasználtuk ebben a lépésben, hogy *foglK³0* és a művelet előtt *UK* igaz volt; emiatt az 1., 2. és 4. tényező igaz kell legyen, azaz elhagyható; a 3. meg egyszerűsíthető. Így kapjuk, hogy

<várj amíg foglK≠0 majd foglK:=foglK+1>

az *UK*-ban a foglNy foglK-val „azonos" szerepű, így levezetés nélkül is tudható az eredmény:

<várj amíg foglNy≠0 majd foglNy:=foglNy+1>

Könnyen látható az eddgiek alapján, hogy feltétel nélkül hajthatók végre a számlálót csökkentő atomi tevékenységek:

< foglK:=foglK-1 >

< foglNy:=foglNy-1 >

L4. Az atomi tevékenységek implementálása:

Az előző modellbeli gondolatot követve most is alkalmaznunk kell a jólbevált *stafétabot továbbadása* módszert. Legyen a **b**, **d1**, **d2**, **d12, dK, dNy** szemaforok rendre a belépéshez, a *(fogl1=0)*, a *(fogl2=0)*, a *(fogl1=0 és fogl2=0)*, ill. a *(foglK=0)*, a *(foglNy=0)* feltételekhez, továbbá a **Db1**, **Db2**, **Db12, ill. DbK, DbNy** a megfelelő számlálók. Így a megoldás:

Változó **fogl1,fogl2,foglK,foglNy: Egész(0)** {UK}
b: BinSzemafor(1)
d1,d2,d12,dK,dNy: BinSzemafor(0) {b+d1+d2+d12∈{0,1}, ui. BinSzemafor hasítás}
Db1,Db2,Db12,DbK,DbNy: Egész(0)

Folyamat KNy(i: 1..DbKNy):
halad(Ny{felé})

P(b) {UK}
Ha foglK≠0 akkor DbK:=DbK+1; V(b); P(dK)
{UK ∧ foglK=0}
foglK:=foglK+1
SIGNAL $_1$

P(b) {UK}
Ha fogl1≠0 akkor Db1:=Db1+1; V(b); P(d1)
{UK ∧ fogl1=0}
fogl1:=fogl1+1
SIGNAL $_2$

P(b) {UK}
foglK:=foglK-1 {UK}
SIGNAL $_3$

keresztez

P(b) {UK}
fogl1:=fogl1-1 {UK}
SIGNAL $_4$

halad(Ny{felé})
Folyamat vége.

Folyamat NyK(i: 1..DbNyK):
halad(K{felé})
P(b) {UK}
Ha foglNy≠0 akkor DbNy:=DbNy+1; V(b); P(dNy)
{UK ∧ foglNy=0}
fogl*Ny*:=fogl*Ny*+1
SIGNAL $_5$

P(b) {UK}
Ha fogl2≠0 akkor Db2:=Db2+1; V(b); P(d2)
{UK ∧ fogl2=0}
fogl2:=fogl2+1
SIGNAL $_6$

P(b) {UK}
foglNy:=foglNy-1 {UK}
SIGNAL $_7$

keresztez

P(b) {UK}
fogl2:=fogl2-1 {UK}
SIGNAL $_8$

halad(K{felé})
Folyamat vége.

Folyamat KD(i: 1..DbKD):
halad(Ny{felé})

P(b) {UK}
Ha foglK≠0 akkor DbK:=DbK+1; V(b); P(dK)
{UK ∧ foglK=0}
foglK:=foglK+1
SIGNAL $_9$

P(b) {UK}
Ha fogl1≠0 vagy fogl2≠0 akkor Db12:=Db12+1; V(b); P(d12)
{UK ∧ fogl1=0 ∧ fogl2=0}
fogl1:=fogl1+1; fogl2:=fogl2+1
SIGNAL$_{10}$
P(b) {UK}
foglK:=foglK-1 {UK}
SIGNAL$_{11}$

keresztez

P(b) {UK}
fogl1:=fogl1-1; fogl2:=fogl2-1 {UK}
SIGNAL$_{12}$

halad(D{felé})
Folyamat vége.

... a folytatás részben mechanikusan kapható meg, részben változatlan az előzőhöz képest ...

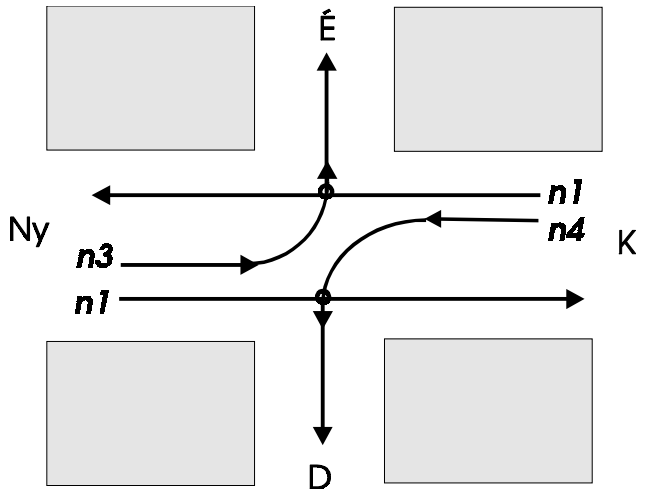
SIGNAL$_i$: Elágazás
fogl1=0 és Db1>0 esetén Db1:=Db1-1; V(d1)
fogl2=0 és Db2>0 esetén Db2:=Db2-1; V(d2)
fogl1=0 és fogl2=0 és Db12>0 esetén Db12:=Db12-1; V(d12)
foglK=0 és DbK>0 esetén DbK:=DbK-1; V(dK)
foglNy=0 és DbNy>0 esetén DbNy:=DbNy-1; V(dNy)
egyéb esetben V(b)
Elágazás vége

6. Útkereszteződés jobbkéz szabállyal

Térjünk vissza ismét az alapfeladat 4-beli megoldása végén említett megjegyzésekhez. Ott szerepelt az a „kifogás”, hogy nemdeterminisztikusan dől el sokszor, hogy pl. egy egyenesen tovább haladó, vagy egy szemből, nagyívben kanyarodó autó veszi-e előbb igénybe az útkereszteződést. Ez a működés eltér a „szabályozott” valóságtól. Szűkítsük a feladat feltételeit az alábbiakkal!

M1. Az egyenesen haladóknak is, és a kanyarodóknak is rendelkezésre áll külön egy-egy sáv (legalábbis a kereszteződéshez közeli útszakaszon).

M2. Továbbá tegyük azt is föl, hogy az elágazásnál a *kanyarodási szabály* érvényesül. A *kanyarodási szabály*: az egyenesen haladóknak előnye van a nagyívben kanyarodóval szemben. Amikor két nagyívben kanyarodó találkozik szembe, akkor ők zavartalanul képesek elhaladni egymás mellett.



Megoldás:

L1. Problémadefiníció:

Az M1 módosítás miatt nem kell ügyelnünk a 4. feladat 3. megjegyzésében feltárt problémára. (Erre koncentráltunk épp az előző feladatban.) Továbbá konfliktushelyzet is csak akkor alakul ki, amikor egy kanyarodó szemben találja magát egy egyenesen szemből jövővel., azaz

1. K→Ny & Ny→É,
2. K→D & Ny→K.

(Ennyiben emlékeztet a 3. és 4. feladatra.) Az újdonságot és a nehézséget épp a M2 módosítás okozza: megkülönböztetendővé téve a konfliktus szereplőit. Emiatt a felállítandó szinkronizációs invariáns elveszti a szerepők szempontjából észrevehető szimmetriáját. (L. a 4. feladat UK-jában a *fogl1*, *fogl2*-t!)

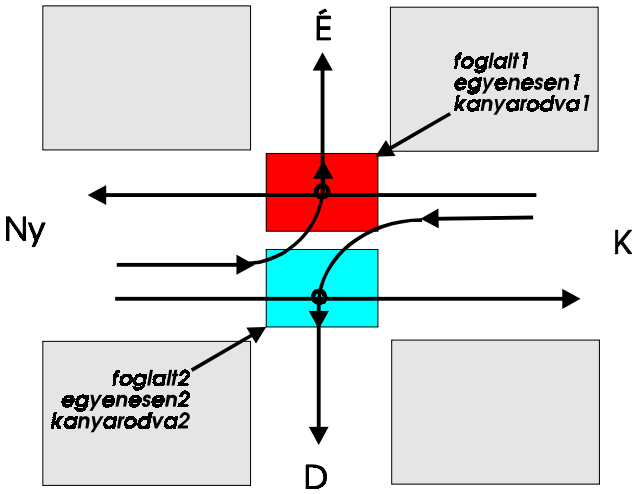
A probléma nem teljesen újkeletű, hiszen hasonlóval kellett megkűzdeni a 2. filozófusos feladatban. Ott az ötlet egy többlet –a szándék kifejezésére vonatkozó– állapot bevezetése volt. Járjunk el most is hasonlóan! Minden autó mielőtt kisajátírhatna egy számára szükséges kereszteződés-sávot, bejelenti igényét rá. Két lehetőség látszik ebben a pillanatban.

a) Az eddigi kereszteződéses feladatmegoldások gondolatmenetét követve, mind a két (K1, K2) kereszteződés-szakaszhoz egy-egy számláló-hármast rendelünk; külön az egyenesen érkező, ill. a kanyarodó igénybejelentőknek, és a már a benntlévőknek. (L. az ábrát.)

b) A filozófusos feladatnál alkalmazott, azonban az eddig már megszokottól alaposan eltérő, módszer: minden egyes autófolyamathoz rendeljünk egy állapotváltozót, amelyek a *halad, igényel, keresztez* érték valamelyikével rendelkezik minden egyes időpillantban.

Kövessük először az első elképzelést, majd tapasztalni fogjuk, hogy zsákutcába vezet az előző rokonfeladatoknál kialakult és megfelelőnek bizonyult „hagyomány”.

a)



Ekkor a megoldásbeli folyamatok algoritmusai számlálónövelő és csökkentő atomi utasításokból kb. így fög fölépülni:

```
Folyamat  xx(i: 1..DbKNy):
    ...
    <egyenesen1:=egyenesen1+1> vagy <kanyarodva1:=kanyarodva1+1>;
    <foglalt1:=foglalt1+1>
    keresztez
    <foglalt1:=foglalt1-1>
    <egyenesen1:=egyenesen1-1> vagy <kanyarodva1:=kanyarodva1-1>;
    ...
Folyamat vége.
```

Gondoljunk előre! A módszer végeredményeként az atomi utasításokhoz autófolyamat-független őrfeltételek jönnek ki, hisz a számlálók a kereszteződéshez, s nem a folyamatokhoz tartoznak:

B(foglalt1,egyenesen1,kanyarodva1).

Tegyük föl, hogy helyesen írja le pl. a NyÉ folyamatbeli

< várj amíg B(foglalt1,egyenesen1,kanyarodva1) majd foglalt1:=foglalt1+1 >

atomi utasítást. Ekkor azonban garantáltan rossz lesz a KNy folyamatbelihez. Ui. érkezzen egyidőben egy-egy KNy és NyÉ autó. Ezek valamilyen sorrendben végrehajtják az egyenesen1, ill. kanyarodva1 számlálókat. (Ebben nyilván egyik sem akadályozhatja meg a másikat.) Mindkettő a „kulcs", foglalt1-növelés atomi utasításnál tart. Természetesen mindketten ugyanazt az őrzőfeltételt értékelik ki és ugyanazt az igaz vagy hamis értéket fogják kapni. Így tehát a kettejük közötti sorrendet nem a prioritás, hanem valami más dönti majd el. A szomorú következtetés: tudnia kell az őrzőfeltételnek, hogy melyik fajta folyamathoz tartozik, s mely folyamatokkal került az adott konfliktushelyzetbe. Ez éppen a b) elképzelés.

b)

$$\mathbf{UK:}\ (\forall i\in[1,4]:\ 0\le n_i)\ \wedge$$
$$((n_1>0 \wedge n_3=0) \wedge (n_1=0 \wedge n_3\ge 0))\ \wedge$$
$$((n_2>0 \wedge n_4=0) \wedge (n_2=0 \wedge n_4\ge 0))$$

L2. Megoldásvázlat:

```
Változó n1,n2,n3,n4: Egész(0)   {UK}

Folyamat  KNy(i: 1..DbKNy):
    halad(Ny{felé})
    <n1:=n1+1>;  keresztez;  <n1:=n1-1>
    halad(Ny{felé})
Folyamat vége.

Folyamat  NyK(i: 1..DbNyK):
    halad(K{felé})
    <n2:=n2+1>;  keresztez;  <n2:=n2-1>
    halad(K{felé})
Folyamat vége.

Folyamat  KD(i: 1..DbKD):
    halad(Ny{felé})
    <n4:=n4+1>;  keresztez;  <n4:=n4-1>
    halad(D{felé})
Folyamat vége.

Folyamat  NyÉ(i: 1..DbKD):
    halad(K{felé})
    <n3:=n3+1>;  keresztez;  <n3:=n3-1>
    halad(É{felé})
Folyamat vége.

... a folytatás a korábbihoz hasonló ...
```

L3. Az invariancia (további) biztosítása:

Vizsgálnunk kell, az alábbi atomi tevékenységek milyen feltétellel biztosítják a *UK* szinkronizációs invariánst:

<n1:=n1+1>, <n2:=n2+1>, <n3:=n3+1>, <n4:=n4+1>,
<n1:=n1-1>, <n2:=n2-1>, <n3:=n3-1>, <n4:=n4-1>

Lf(<n1:=n1+1>, UK) ≡

≡ Lf(<n1:=n1+1>, $(\forall i\in[1,4]:\ 0\le n_i)\ \wedge$
$((n_1>0 \wedge n_3=0) \vee (n_1=0 \wedge n_3\ge 0)) \wedge ((n_2>0 \wedge n_4=0) \vee (n_2=0 \wedge n_4\ge 0)))$ ≡

≡ $(\forall i\in[1,4]:\ 0\le n_i)\ \wedge$ (1)
$((n_1+1>0 \wedge n_3=0) \vee (n_1+1=0 \wedge n_3\ge 0)) \wedge ((n_2>0 \wedge n_4=0) \vee (n_2=0 \wedge n_4\ge 0))$ ≡

≡ $((n_3=0)) \wedge ((n_2>0 \wedge n_4=0) \vee (n_2=0 \wedge n_4\ge 0))$ ≡ (2)

≡ $n_3=0$

Az (1)-nél az első logikai tényezőt, mint triviálisan igazat, elhagytuk, továbbá a második tényező első tagját redukáltuk, a második tagja $(n_1+1=0 \wedge n_3\ge 0)$ azonosan hamis az *UK* igaz volta mellett, így elhagyható. A (2)-nél az *UK* korábbi teljesülését elegendő figyelembe venni. Így kapjuk, hogy

<várj amíg n3≠0 majd n1:=n1+1>

hasonlóan kapható, hogy

<várj amíg n4≠0 majd n2:=n2+1>

<várj amíg n1≠0 majd n3:=n3+1>

<várj amíg n2≠0 majd n4:=n4+1>

Másrészt

Lf(<n1:=n1-1>, UK) ≡

≡ Lf(<n1:=n1-1>, $(\forall i\in[1,4]:\ 0\le n_i)\ \wedge$
$((n_1>0 \wedge n_3=0) \vee (n_1=0 \wedge n_3\ge 0)) \wedge ((n_2>0 \wedge n_4=0) \vee (n_2=0 \wedge n_4\ge 0)))$ ≡

≡ $(\forall i\in[1,4]:\ 0\le n_i)\ \wedge$ (1)
$((n_1-1>0 \wedge n_3=0) \vee (n_1-1=0 \wedge n_3\ge 0)) \wedge ((n_2>0 \wedge n_4=0) \vee (n_2=0 \wedge n_4\ge 0))$ ≡

≡ $((n_1>1 \wedge n_3=0) \vee (n_1=1 \wedge n_3\ge 0)) \wedge ((n_2>0 \wedge n_4=0) \vee (n_2=0 \wedge n_4\ge 0))$ ≡ (2)

≡ Igaz

Mind az (1), mind a (2) lépésben kihasználtuk az *UK* invarianciáját, ill. azt szituációt (ti. hogy egy <n1:=n1+1> művelet utáni állapotban van a folyamat), amelyben az érintett atomi műveletre sor került. Hasonló levezetési lépések után kapjuk mind a négy csökkentő atomi értékadásra, hogy őrző feltétel nélkül alkalmazható:

< n1:=n1-1 >

< n2:=n2-1 >

< n3:=n3-1 >

< n4:=n4-1 >

L4. Az atomi tevékenységek implementálása:

A már jólbevált módszerrel kapjuk meg a program implementációját. Be kell vezetnünk szemaforokat a belépéshez, s az örző feltételekhez (*(n1=0)*, *(n2=0), (n3=0)*, ill. *(n4=0)*), s számlálókat. Ezeket rendre jelöljük a korábbi programhoz hasonlóan: **b**, **d1**, **d2**, **d3**, **d4**; **Db1**, **Db2, Db3**, **Db4**.

Változó **n1,nl2,n3,n4: Egész(0)** {UK}
b: BinSzemafor(1)
d1,d2, d1, d2: BinSzemafor(0) {b+d1+d2∈{0,1}, ui. BinSzemafor hasítás}
Db1,Db2,Db3,Db4: Egész(0)

Folyamat KNy(i: 1..DbKNy):
halad(Ny{felé})

P(b) {UK}
Ha n3≠0 akkor Db1:=Db1+1; V(b); P(d1)
{UK ∧ n3=0}
n1:=n1+1
SIGNAL $_1$

keresztez

P(b) {UK}
n1:=n1-1 {UK}
SIGNAL $_2$

halad(Ny{felé})
Folyamat vége.

Folyamat NyK(i: 1..DbNyK):
halad(K{felé})

P(b) {UK}
Ha n4≠0 akkor Db2:=Db2+1; V(b); P(d2)
{UK ∧ n4=0}
n2:=n2+1
SIGNAL $_3$

keresztez

P(b) {UK}
n2:=n2-1 {UK}
SIGNAL $_4$

halad(K{felé})
Folyamat vége.

Folyamat KD(i: 1..DbKD):
halad(Ny{felé})

P(b) {UK}
Ha n2≠0 akkor Db4:=Db4+1; V(b); P(d4)
{UK ∧ n2=0}
n4:=n4+1
SIGNAL $_5$

keresztez

P(b) {UK}
n4:=n4-1 {UK}
SIGNAL $_6$

halad(D{felé})
Folyamat vége.

... a folytatás részben mechanikusan kapható meg, részben változatlan az előzőhöz képest ...

SIGNAL$_i$: ***Elágazás***
n1=0 és Db1>0 esetén Db1:=Db1-1; V(d1)
n2=0 és Db2>0 esetén Db2:=Db2-1; V(d2)
n1=0 és Db3>0 esetén Db3:=Db3-1; V(d3)
n4=0 és Db4>0 esetén Db4:=Db4-1; V(d4)
egyéb esetben V(b)
Elágazás vége

A feladatsor befejezéseként egy feladat: gondoljuk meg, hogy mit jelentene az útkereszteződéses feladat olyan bővítése, amelyben megengedett a kisívben történő kanyarodás is!

Implementációk

Ebben a fejezetben a párhuzamos végrehajtás konkrét nyelvi környezetben történő megvalósításának problémáit és ezek feloldásait tekintjük át. Először érdemes a „klasszikus” nyelvek egy fajtáját szemügyre venni e szempontból. Választásunk a módszertani és elterjedtség szempontjából ideálisnak mondható Pascal nyelv változatára esett: a Turbo Pascal 7.0.

1. Egy szekvenciális –Turbo Pascal 7.0-nyelvű megoldás

1.1. Kontrollmentes párhuzamos m űködés

Az első tanulságos példa: egy –úgy is mondhatnánk– *rossz példa* lesz. Ui. arról szól, hogy a párhuzamosan zajló folyamatok mindenfajta kontrollálása nélküli megvalósítás gyakran jut *holtpont*ra. Ebben a programpéldában ezt is szemügyre vehetjük, de ami legalább ennyire érdekes, az a szekvenciális programkörnyezetbeli megszervezése a párhuzamosságnak.

Nézzük elképzelésünket pontokba szedve:

1. *Minden folyamat*ot egy időegység alatt végrehajtandó *lépések sorozatá*ra bontjuk. Ezek alatt nem kommunikálhatnak egymással. Ez az a feltevés, ami éppen a *szinkronizáció hiányá*t jelenti. A lépések kódrészeit elkülönítve tartalmazza a 'lepes' metódus. A lépések végrehajtási sorrendje és a leírási sorrendje a 'lepes' metódusban megegyezik. (Persze semmi akadálya nincs annak, hogy a lépéseket ciklusokba zárjuk, vagy feltételekkel bővítsük ki.)
2. Az egyes folyamatok vezérlését a 'keret' metódus végzi, amely meghívja a 'lepes' metódust a megfelelő lépéssorszámmal.
3. Hogy éppen melyik lépésnél tart, azt egy attributuma (az 'állapot' mezője) tartja nyilván. Mint egy *programszámláló*t képzelhetjük el, külön-külön minden egyes folyamathoz. Az állapotok, vagyis a lépések számát rögzíti a 'VegAllapot' konstans.
4. Feltesszük, hogy a folyamatok *egyetlen folyamat példányai*. (Bár nem lenne elvi akadálya annak sem, hogy különböző folyamatok szerepeljenek a programban. Csak a kód hosszabbodna meg, lényeges többletmondanivaló nélkül.)
5. A teljes rendszer vezérlését (a főfolyamat munkáját) a *főprogram* végzi, amely időegységenként *minden folyamatnak átadja egy lépés erejéig a vezérlést*. Ő gondoskodik az új folyamatok *létrehozásá*ról, s itt *szűnnek meg* a végállapotba jutottak.

```
Program FolyamatSzimulacio;
{                    Párhuzamos folyamatok kvázipárhuzamos szimulációja.                    }
{******* Folyamat-típus ***************************************************}
   Const
      VegAllapot = konkrét érték;        {Az egyes folyamatok lépéseinek, azaz állapotainak száma.}
```

```
Type
   Lepesek  = 1..VegAllapot;
   Folyamat = Object
        azon: Integer;    {folyamatazonosító}
        allapot: Byte;
        Constructor indul(Const fa: Byte);
        Procedure keret;
        Procedure lepes(Const i: Lepesek);
   End;
   Constructor Folyamat.indul(Const fa: Byte);
   Begin
        azon:=fa;  allapot:=1; {minden folyamat az 1-s állapotból indul}
   End;

   Procedure Folyamat.keret;          {az adott folyamat egy –a következő– lépésének vezérlése}
   Begin
        Lepes(allapot);  Inc(allapot);
   End;

   Procedure Folyamat.lepes(Const i: Lepesek);
   Begin
        Case i of
            1: {az 1. lépés kódja};
            2: {a  2. lépés kódja};
            {... a további lépések ágai ...}
        End;
   End;

{******* Folyamat-típus ****************************************************}

Const
   MaxN = 10;                         {Maximum ennyi folyamat lehet egyszerre a rendszerben.}

Type
   Folyamatok = Array [1..MaxN] of Folyamat;
Var
        {a rendszer állapotparaméterei:}
   F : Folyamatok;
   N : Byte;             {aktív folyamatok száma}
   i,T : Integer;
        {a rendszer bemenőparaméterei:}
   P : Real;             {új folyamat születésének valószínüsége}
   SzulVarSzam: Byte; {az egyszerre születő folyamatok várható száma}
        {a rendszer kimenőparamétere:}
   DbF: Integer;       {összesen generált folyamatok száma}

Procedure General(Var fk: Folyamatok; Var n: Byte; Const db: Byte);
   Var
        i: Byte;
Begin
   i:=1;
   While (i<=db) and (n+i<MaxN) do
   Begin
        Inc(DbF); fk[n+i].indul(DbF); Inc(i);
   End;
   n:=n+i-1;
End;
```

```
    Procedure Takarit(Var fk: Folyamatok; Var n: Byte);
        Var
            i,db: Byte;
    Begin
        db:=0;
        For i:=1 to n do
        Begin
            If fk[i].allapot=VegAllapot then fk[i].vege
                                        else Begin Inc(db); fk[db]:=fk[i]; End;
        End;
        n:=db;
    End;
Begin {a rendszer (=folyamatok együttese) kvázipárhuzamos vezérlése:}
    P:=0.5; SzulVarSzam:=10;
    DbF:=0;
    T:=1; General(F,N,Random(SzulVarSzam)+1{legalább 1 folyamattal indul});
    While N>0 do
    Begin
        For i:=1 to N do F[i].keret;
        Takarit(F,N);

        If Random<P then Begin General(F,N,Random(SzulVarSzam)); End;
        Inc(T);
    End;
End.
```

Ezt az általános keretet használhatjuk föl pl. az (előző fejezetben elsőként tárgyalt) útkereszteződéses feladat megoldásának kipróbálásra. Annyi módosítással, hogy megengedjük a kisívben történő kanyarodást is.

Most nem kell többet tennünk, mint a feladat konkrétumait beilleszteni a fenti keretbe. Összefoglaljuk, miket is várunk el a programunktól:

A kereszteződés:

1. A főútvonal 2 irányban 1-sávos. (Előzés nincs.)
2. A kereszteződésnél a főútról csak le lehet kanyarodni, rá nem.
3. A mellékút 1-sávos.
4. Megengedett a főútról való balra és jobbra kanyarodás, ill. egyenesen továbbhaladás.

A modell:

1. *Nincs szinkronizáció*, de ütközés sem.
2. Generálás: időegységenként valahány (paramétertől függő) darab.
3. A célkód:

   K→Ny =+1..+99; K→E =+101..+199; K→D =+201..+299
   Ny→K =-1..-99; Ny→E =-101..-199; Ny→D =-201..-299

4. Az autóknak *3 állapota* van: a kereszteződés előtti, a kereszteződésbeli, ill. az utáni. (Nyilván semmi akadálya nincs annak, hogy az előtti és utáni szakaszt „finomabban” bontsuk föl, a párhuzamosság elvi szempontjából azonban semmi többlet információt nem nyújt.)
5. A könnyebb adminisztrálás (az ütközés elkerülése) érdekében *följegyezzük*, hogy a kereszteződés adott irányú *sáv*ja *foglalt-e*. (Ezt két logikai változó tartalmazza majd.)
6. Foglalt sáv irányába nem lehet elfordulni.

Várható eredmény:

Előbb-utóbb *halálos ölelés*.

```
Program FolyamatSzimulacio_UtKeresztezodes1;
{                Párhuzamos folyamatok kvázipárhuzamos szimulációja – útkereszteződés..          }
   ...
   Const
      VegAllapot = 3; {állapotok: 1='a kereszteződés előtt',
                                  2='a kereszteződésben',
                                  3='a kereszteződés után'}
   ...
```

A feladathoz való jobb „illeszkedés” kedvéért Folyamat helyett Auto-t mondunk, s a metódusokat is, ha kell, új tartalommal töltjük meg.

Az ütközések elkerülése érdekében (s nem pedig bármifajta párhuzamossági szinkronizálás miatt!) megjegyezzük azt, ha kereszteződés egyik, ill. másik sávját egy autó elfoglalta. Ezt két logikai változó a: '**fogl1**', ill. '**fogl2**' rögzíti.

```
   Var
      fogl1,fogl2: Boolean; {K->Ny, ill.Ny->K sáv foglalt}
{******* Autó-típus ******************************************************}
   Type
      Auto =  Object(Folyamat)
                  ...
      End;
      Constructor Auto.indul(Const fa: Byte);
      Begin
          azon:=(Integer(Random(2))*2-1)*(Random(3)*100+fa); {utolsó 2 számjegy egyedi}
          allapot:=1;
      End;
      Procedure Auto.keret; {az adott folyamat egy – a következő- lépésének vezérlése}
      Begin
          Lepes(allapot);  Inc(allapot);[5]
      End;
```

Az ütközés elkerülendő, ami a 'lepes' metódus feladata figyelemmel kísérni.

[5] Ingerlő azonosság miatt hajlamosak lennénk az ős osztály 'keret' metódusát hívni: *Inherited keret*, de mivel az esetben az ahhoz tartozó 'lepes' lenne aktivizálva, ami viszont eltér az 'Auto' esetén alkalmazandótól, inkább megismételjük a kódot.

```
    Procedure Auto.lepes(Const i: Lepesek);
    Begin
        Case i of
          1: {az 1. lépés kódja:}
             Begin
                If
                  ({K->Ny}(azon>0) and (azon<100) and fogl1) or
                  ({K->E }(azon>100) and (azon<200) and fogl1) or
                  ({K->D }(azon>200) and fogl1) or
                  ({Ny->K}(-azon>0) and (-azon<100) and fogl2) or
                  ({Ny->E}(-azon>100) and (-azon<200) and fogl2) or
                  {Ny->D}(-azon>200) and fogl2
                                          then Dec(allapot) {nem léphet} else if
                  {beléphet K-ről} azon>0 then fogl1:=True
                  {beléphet Ny-ról}          else fogl2:=True
                {EndIf};
             End;
          2: {a  2. lépés kódja:}
             Begin
                If
                  ({K->D }(azon>200) and fogl2) or
                  ({Ny->E}(-azon>100) and (-azon<200) and fogl1)
                                       then Dec(allapot) {nem léphet} else if
                  {K-ről jött} azon>0 then fogl1:=False {kiléphet}
                  {Ny-ról jött}          else fogl2:=False {kiléphet}
                {EndIf};
             End;
        End{Case};
    End;
{******* Autó-típus *****************************************************}
```

'Folyamatok' helyett persze 'Autok' tömbünk lesz.

```
  Const
     MaxN  = 10;
  Type
     Autok = Array [1..MaxN] of Auto;
  Var
              {a rendszer állapotparaméterei:}
     F : Autok;
     ...
  Procedure General(Var fk: Autok; Var n: Byte; Const db: Byte);
     ...
  Procedure Takarit(Var fk: Autok; Var n: Byte);

     ...
```

Némi –algoritmikai szempontból– lényegtelen módosítással a főprogram:

```
Begin {a rendszer (=folyamatok együttese) kvázipárhuzamos vezérlése:}
   P:=1;  SzulVarSzam:=10;  fogl1:=False;  fogl2:=False;
   DbF:=0;
   T:=1; General(F,N,Random(SzulVarSzam)+2{legalább 2 folyamattal indul});
      ...
End.
```

Egy futás „históriáját” adjuk meg az alábbi táblázatban.

Idő	Folyamat-azonosító	Folyamat-lépésszám
1:	1	1
	-2	1
2:	1	2
	-2	2
	-3	1
	-104	1
	-105	1
3:	-3	2
	-104	1
	-105	1
4:	-104	2
	-105	1
	106	1
	7	1
	208	1
	-9	1
	-10	1
	11	1
	-12	1
5:	-105	2
	106	2
	7	1
	208	1
	-9	1
	-10	1
	11	1
	-12	1
	13	1

Idő	Folyamat-azonosító	Folyamat-lépésszám
6:	-105	2
	7	2
	208	1
	-9	1
	-10	1
	11	1
	-12	1
	13	1
	214	1
7:	-105	2
	208	2
	-9	1
	-10	1
	11	1
	-12	1
	13	1
	214	1
	15	1
8:	-105	2
	208	2
	-9	1
	-10	1
	11	1
	-12	1
	13	1
	214	1
	15	1

Mint látható a 7. időegységtől kezdődően a rendszer állapota befagy. Ui. az 5. autó, amely nyugatról északra nagyívben (-105), a 8., amely pedig keletről délre akar –szintén– nagyívben kanyarodni, egymásra várva állják el a másik útját a kereszteződésbeli saját sávjukban. Tökéletes holtpont helyzet.

1.2. Kontrollált párhuzamos működés – egy szemaforos változat

Az útkereszteződéses probléma a **változók cseréje** módszer szerinti megoldását implementáljuk most. Megfigyelhető lesz, hogy miként lehet szemaforokat kezelni és ezt beilleszteni a kvázipárhuzamos működésű programba.

Kezdjük az elképzelés vázolásával, de csak az előzőekben már leírtakat egészítjük ki, vagy módosítjuk alább.

ad 1. A kommunikáció tiltása természetesen csak a szemaforműveletekre korlátozódik.

ad 5. ... Az *inaktív* –valamely szemafornál várakozó– folyamatok közül csak az *első*vel történhet valami (továbbléphet, vagy még várakozni kényszerül).

6. A *szemaforműveletek* önálló, elemi műveleteknek számítanak. A szemafor *P*-müveletét két műveletre szedjük:

   a) a „normál” *P*-művelet, ami adott esetben a *várakozási sorba állít*; és

   b) a *Q*-művelet, amely *ellenőrzi* a várakozást, s adott esetben *továbbengedi*. (Ez utóbbi nem tűnik föl az egyedi folyamat lépései között, csak a főfolyamatban.)

A kereszteződés:

... ugyanaz, mint az előbb volt ...

A modell:

1. Egyszerű, *egy-szemaforos szinkronizáció*.
2. Generálás: időegységenként valahány (paramétertől függő) darab.
3. A célkód:

   ... ugyanaz, mint volt az előbb ...

4. A kereszteződésben *csak egy* autó lehet. A belépést a *szemafor*ral szinkronizáljuk.
5. Az autóknak 5 állapota van: a kereszteződés előtti, a szemaforállító P-művelet végrehajtása alatti, a kereszteződésbeli, a szemaforállító V-művelet végrehajtása alatti, ill. a kereszteződés utáni. (Nyilván most sincs semmi akadálya nincs annak, hogy az előtti és utáni szakaszt „finomabban” bontsuk föl.)

Várható eredmény:

Nincs halálos ölelés, de meglehetősen „darabos” az autók áramlása (a modell 4. pontja miatt).

```
Program FolyamatSzimulacio_UtKeresztezodes2;
                    {   Párhuzamos folyamatok kvázipárhuzamos szimulációja,
                                szemaforos szinkronizációval.          }
{******* Folyamat-típus ****************************************************}
   Const
      VegAllapot = 5; {állapotok: 1='a kereszteződés előtt',
                                  2='P-művelet',
                                  3='a kereszteződésben',
                                  4='V-művelet',
                                  5='a kereszteződés után'}
```

A Pascal programban egy osztályhierarchiát építünk föl. A legalapvetőbb osztály maga a *Folyamat* osztály, amely lényegében megegyezik az előző implementációbelivel. Majd, mivel a szemaforok egy-egy önálló sort kezelnek, célszerű definiálni egy általános *Sor* osztályt, amit fölhasználva definiáljuk a kellő konkrét sorokat. Mivel –sajnos– a Pascal osztályai nem parametrizálhatók (nem ismeri a *generic* fogalmát), ezért a *Sor* osztály most *Integer* típusú elemek kezelésére lesz alkalmas. (Ez által a sorba tehetők a folyamatazonosítók.)

```
 Type
      FolAzo   = Integer;        {folyamatazonosító}
      Lepesek = 1..VegAllapot;
      Folyamat=Object
            azonosito: FolAzo;
            allapot: Byte;
            Constructor indul(Const fa:Integer);
            Procedure keret;
            Procedure lepes(Const i:Lepesek);
      End;
{******* Folyamat-típus ****************************************************
```

```
******* Sor-típus ************************************************************}
  Const
     MaxSor = 10;
  Type
     Ido=Integer;
     Sor=Object{Generic=FolAzo}
          hossz,                {sorbeli elem száma}
          eleje,                {az első, sorbeli elem indexe}
          vege : -1..MaxSor;    {az utolsó, sorbeli elem indexe}
          elem : Array [0..MaxSor-1] of FolAzo;
          hiba : Boolean;       {az utolsó sorművelet sikeres volt-e?}
          Constructor inic;
          Procedure Sorba(Const o:FolAzo);
          Procedure Sorbol(Var o:FolAzo);
          Procedure Elso(Var el:FolAzo);
          Function SorHossz:Integer;
     End;
{******* Sor-típus ************************************************************
```

A későbbi általánosíthatóság miatt definiáljuk az *IdőFolyamatSor* osztályt, amely a folyamatazonosító és (esemény-) idő kettősét tárolja egy sorelemként. Az asszociált műveletek paramétereiben akkurátusan megőrizzük a kétféle, bár „alakilag" azonos, Integer paraméter típusát.

```
******* IdoFolyamatSor-típus ************************************************}
  Type
     IdoFolyamatSor=Object
          ido: Sor{Ido};
          fol: Sor{FolAzo};
          Constructor inic;
          Procedure Sorba(Const i:Ido; Const o:FolAzo);
          Procedure Sorbol(Var i:Ido; Var o:FolAzo);
          Procedure Elso(Var i:Ido; Var o:FolAzo);
          Function SorHossz:Integer;
     End;
{******* IdoFolyamatSor-típus ************************************************
```

A *Szemafor* osztály fölhasználja az imént definiált osztályt, mint a *várakozó folyamat*okat tároló mezőjének típusát. Megjegyezzük, hogy a *Szemafor*okhoz hozzárendelünk egy függvényt (*SorHossz*) is, amihez hasonlónak a „praktikusságát" a korábbi elméleti részben már előre jeleztük.

```
******* Szemafor-típus ******************************************************}
  Type
     Szemafor=Object{Generic=Folyamat}
          sz: Integer;
          varoSor: IdoFolyamatSor;        {az inaktív folyamatok sora: idő + folyamatazonosító}
          Constructor inic;
          Procedure P(Const t:Ido; Var o:Folyamat);
                          {ha a szemafor=0, akkor a paraméter Folyamatot a varoSor-ba
                           teszi, s o.allapot visszalép}
          Procedure Q(Var o:Folyamat);
                          {ha a szemafor>0, akkor a varoSor-beli első Folyamatot adja vissza,
                           egyébként „üres-Folyamatot"}
          Procedure V(Const o:Folyamat);
          Function SorHossz:Integer;
     End;
{******* Szemafor-típus ******************************************************
```

Az *aktív* folyamatok kezelését a *PFolyamat* osztállyal definiáljuk. Ebbe illesztettük bele –az előző implementációtól eltérően– a szimulációs időt, s az aktív folyamatok (idő-folyamat-)sorát.

```
******* PFolyamat-típus ****************************************************}
  Type
    PFolyamat=Object{Generic=Folyamat}
        T  : Word;            {akt. idő}
        aktivSor: IdoFolyamatSor; {az aktív folyamatok sora: idő + folyamatazonosító}
        Constructor inic;
        Procedure FolyamatSzul;   {folyamatot generál és tesz az aktívSor utolsó elemévé}
        Procedure MelyFolyamat(Var f:Folyamat; Var j:Integer);
                              {f.azonosito=>f=ok[j] (ok-tömbből)}
        Function FolyamatValtozott(Const f:Folyamat; j:Integer):Boolean;
        Procedure FolyamatModosit(Const f:Folyamat; j:Integer);
                              {ok[j]:=f}
        Procedure FolyamatTemet(Const i:Integer);
        Procedure Sorba(Const i:Ido; Const f:Folyamat);
        Procedure Sorbol(Var i:Ido; Var f:Folyamat);
        Procedure Elso(Var i:Ido; Var f:Folyamat);
        Function SorHossz:Integer;
    End;
{******* PFolyamat-típus ****************************************************}

  Const
    UresFolyamat : Folyamat=(azonosito:-MaxInt);
    MaxObj = 1000;
```

Az alábbiak a Pascal „szerencsétlenségei” miatt kerültek ide. Az *ok*, *oDb*, *oUt* a későbbi *PFolyamat*-hoz tartoznak ugyan, de mivel a *Szemafor* is használja, ezért ide, előre, globális változóként kerültek.

```
  Type
    Folyamatok = Array [1..MaxObj] of Folyamat;
  Var
    ok   : Folyamatok;        {az összes folyamat tömbje}
    oDb: 0..MaxObj;           {az összes folyamat darabszáma}
    oUt : 0..MaxObj;          {az összes folyamat utolsója}
  Var
    pf: PFolyamat;    f : Folyamat;     s : Szemafor;
    P : Real;         i,j,t,SzulVarSzam: Integer;

    Constructor Folyamat.indul(Const fa:Integer);
    Begin
        azonosito:=fa;  allapot:=1;
    End;

    Procedure Folyamat.keret;        {az adott folyamat egy – a következő- lépésének vezérlése}
    Begin
        Lepes(allapot);  Inc(allapot);
    End;
```

Az világos, hogy a *Folyamat.lepes* metódus tartalmazza azokat a konkrét ismereteket, amitől ez a program épp a konkrét, útkereszteződéses feladathoz tartozik. Az elemi lépésekre bontást kellene újra írni egy másik feladat esetén.

```
    Procedure Folyamat.lepes(Const i:Lepesek);
    Begin
        Case i of
          1:  Begin {az 1. lépés kódja}
              End;
          2:  Begin {a  2. lépés kódja}
                  s.P(pf.T,self);
              End;
          3:  Begin {az 3. lépés kódja}
                  s.V(self);
              End;
```

```
          4:  Begin {a  4. lépés kódja}
              End;
      End;
End;
Constructor Sor.inic;
Begin
      hossz:=0;  eleje:=0;  vege:=-1;  hiba:=False
End{Sor.inic};

Procedure Sor.Sorba(Const o:FolAzo);
Begin
      If hossz<MaxSor then
      Begin
         Inc(hossz); vege:=(vege+1) Mod MaxSor; elem[vege]:=o
      End
           Else
      Begin
         hiba:=True
      End;
End{Sor.Sorba};

Procedure Sor.Sorbol(Var o:FolAzo);
Begin
      If hossz>0 then
      Begin
         Dec(hossz); o:=elem[eleje]; eleje:=(eleje+1) Mod MaxSor;
      End
           Else
      Begin
         hiba:=True
      End;
End{Sor.Sorbol};

Procedure Sor.Elso(Var el:FolAzo);
Begin
      If hossz>0 then el:=elem[0]
                 else hiba:=True
End{Sor.Elso};

Function Sor.SorHossz:Integer;
Begin
      SorHossz:=hossz
End{Sor.SorHossz};

Constructor IdoFolyamatSor.inic;
Begin
      ido.inic; fol.inic;
End{IdoFolyamatSor.inic};

Procedure IdoFolyamatSor.Sorba(Const i:Ido; Const o:FolAzo);
Begin
      ido.Sorba(i); fol.Sorba(o);
End{IdoFolyamatSor.Sorba};

Procedure IdoFolyamatSor.Sorbol(Var i:Ido; Var o:FolAzo);
Begin
      ido.Sorbol(i); fol.Sorbol(o);
End{IdoFolyamatSor.Sorbol};

Procedure IdoFolyamatSor.Elso(Var i:Ido; Var o:FolAzo);
Begin
      ido.Elso(i); fol.Elso(o);
End{IdoFolyamatSor.Elso};
```

```
Function IdoFolyamatSor.SorHossz:Integer;
Begin
    SorHossz:=fol.SorHossz
End{IdoFolyamSor.SorHossz};
Constructor Szemafor.inic;
Begin
    sz:=1;  varoSor.inic;
End{Szemafor.inic};

Procedure Szemafor.P(Const t:Ido; Var o:Folyamat);
Begin
    If sz=0 then
    Begin
        varoSor.Sorba(t,o.azonosito);  Dec(o.allapot)
    End
          Else
    Begin
        sz:=0
    End;
End{Szemafor.P};

Procedure Szemafor.Q(Var o:Folyamat);
    Var
        t:Ido;
        j:Integer;

Begin
    If sz=1 then
    Begin
        varoSor.Sorbol(t{nem fontos},o.azonosito);
      {kikerül, s átkerülhet az aktív folyamatok sorába}
        pf.MelyFolyamat(o,j{nem fontos});
    End
        Else
    Begin
        o.azonosito:=UresFolyamat.azonosito
    End;
End{Szemafor.Q};

Procedure Szemafor.V;
Begin
    sz:=1
End{Szemafor.V};

Function Szemafor.SorHossz:Integer;
Begin
    SorHossz:=varoSor.SorHossz
End{Szemafor.SorHossz};

Constructor PFolyamat.inic;
Begin
    aktivSor.inic;
    oDb:=0;  {eddig generált folyamatok száma=0}
    oUt:=0;   {eddig generált folyamatok utolsója a „0”. }
End{PFolyamat.inic};

Procedure PFolyamat.FolyamatSzul;
Begin
    Inc(oDb);
    Inc(oUt); ok[oUt].indul((Integer(Random(2))*2-1)*(Random(3)*100+oDb));
    aktivSor.Sorba(T,ok[oUt].azonosito);
    {uf: minden generált elem egyedi azonosítóval rendelkezik}
End{PFolyamat.FolyamatSzul};
```

```
Procedure PFolyamat.MelyFolyamat(Var f:Folyamat; Var j:Integer);
                                                    {f.azonosito=>f=ok[j] (ok-tömbből)}
Begin
    {ef: van keresett azonosítójú elem}
    j:=1;
    While (ok[j].azonosito<>f.azonosito) do Inc(j);
    f.allapot:=ok[j].allapot
End{PFolyamat.MelyikFolyamat};
Procedure PFolyamat.FolyamatModosit(Const f:Folyamat; j:Integer);
                                                                    {ok[j]:=f}
Begin
    ok[j]:=f;
End{PFolyamat.FolyamatModosit};

Function PFolyamat.FolyamatValtozott(Const f:Folyamat; j:Integer):Boolean;
Begin
    FolyamatValtozott:=f.allapot<>ok[j].allapot;
End{PFolyamat.FolyamatValtozott};

Procedure PFolyamat.FolyamatTemet(Const i:Integer);
    Var
      j:Integer;
Begin
    For j:=i+1 to oDb do ok[j-1]:=ok[j];
    Dec(oUt);
End{PFolyamat.FolyamatTemet};

Procedure PFolyamat.Sorba(Const i:Ido; Const f:Folyamat);
Begin
    aktivSor.Sorba(i,f.azonosito);
End{PFolyamatSor.Sorba};

Procedure PFolyamat.Sorbol(Var i:Ido; Var f:Folyamat);
Begin
    aktivSor.Sorbol(i,f.azonosito);  pf.MelyFolyamat(f,j{nem fontos});
End{PFolyamatSor.Sorbol};

Procedure PFolyamat.Elso(Var i:Ido; Var f:Folyamat);
Begin
    aktivSor.Elso(i,f.azonosito);  pf.MelyFolyamat(f,j{nem fontos});
End{PFolyamatSor.Elso};

Function PFolyamat.SorHossz:Integer;
Begin
    SorHossz:=aktivSor.SorHossz
End{PFolyamat.SorHossz};
```

Elérkeztünk a program „vezérlő részéhez”, amelynek rövid lényege a következő: az inicializáló és néhány a kezdő pillanatban a rendszerbe lépő autó generálása után a szimuláció „életciklusa” jön. Az egész addig folyik míg a rendszerben tartózkodik legalább egy autó, legyen az akár aktív, vagy akár inaktív állapotban. Minden egyes időegységben először az aktív autókkal történik valami állapotváltozás, majd a rendszerben levő szemaforoknál –jelen esetben: az egyetlennél– várakozó autók közül az első próbál onnan kijutni (a Q-művelet segítségével). Az időegység végén „szokásos” dolgokat kell végeznünk: a végállapotba jutott autókat a rendszerből el kell távolítani, s gondoskodni kell esetleges új belépőkről.

```
Begin
   P:=0.5; SzulVarSzam:=1;
   pf.inic; s.inic;  pf.T:=1;
   For i:=1 to SzulVarSzam+2{legalább 2 folyamattal indul} do
      If Random<P then pf.FolyamatSzul;
```

```
    While (pf.SorHossz>0) or (s.SorHossz>0) do
    Begin
     {az aktív folyamatok szimulációja:}
       For i:=1 to pf.Sorhossz do
       Begin
            pf.Sorbol(t,f);
            f.keret;  {egyedi folyamat-szimuláció}
            If pf.FolyamatValtozott(f,j) then
            Begin
                pf.FolyamatModosit(f,j);  pf.Sorba(t,f);
            End;
        {Történt-e változás a szemaforok körül?
         Az inaktív –a várakozási sor(ok)ban álló– folyamatok szimulációja (azaz minden
         szemaforra):}
            If s.SorHossz>0 then
            Begin
                s.Q(f);
                If f.azonosito<>UresFolyamat.azonosito then
                Begin  {továbblép:}
                    pf.MelyFolyamat(f,j);
                    f.keret;
                    pf.FolyamatModosit(f,j); pf.Sorba(T,f)
                End;
            End;
       End;
     {a végállapotba jutott folyamatok kihagyása:}
       For i:=1 to pf.SorHossz do
       Begin
            pf.Sorbol(t,f);  pf.MelyFolyamat(f,j);
            If f.allapot=VegAllapot then pf.FolyamatTemet(j)
                                    else pf.Sorba(t,f);
       End;
     {új folyamatok beléptetése:}
       For i:=1 to SzulVarSzam do If Random<P then pf.FolyamatSzul;
     {múlik az idő:}
       Inc(pf.T);
    End;
End.
```

Vissza az elméleti részhez: SzinkronElm.rtf/pdf.

Függelék – a szemaforok anatómiája,
a szemaforműködés egy lehetséges mechanizmusa

A bináris szemaforral szemben a következő elvárásokat fogalmaztuk meg:

1. Adminisztrálja a hozzá elérkező folyamatokat, s csak akkor engedje tovább, amikor nincs a kritikus szakaszban egyetlen folyamat sem (P-művelet).

2. Amikor megakadályozza egy folyamat továbbhaladását, akkor gondoskodjon annak egy várakozási sorba történő állításáról (P-művelet).

3. Amikor vannak várakozó folyamatok és a kritikus szakaszt elhagyja a folyamat, akkor léptesse be a sorelsőt a kritkus szakaszba a továbbiak ismételt várakoztatása mellett (V-művelet).

4. Mind a P-, mind a V-művelet egyszerre csak egy példányban hajtható végre. (Bár az 1.,2.,3. teljesülése eleve kizárja a V-művelet többszörös alkalmazását.)

Annyi világos, hogy a kritikus szakasz kézben tartása érdekében valahogyan adminisztrálni kell a „szemafor állását". Például így:

```
Típus SzemaforÁllás = (Szabad, Foglalt)
      Szamafor = Rekord
                    ( állás: SzemaforÁllás
                      ... )
```

A dolog nem tűnik különösen problematikusnak mindaddig, amíg egy szélsőséges eset előfordulására nem gondolunk. Ez az eset akkor lép föl, amikor „abszolút" *párhuzamosan* érkezik a szemaforhoz több folyamat. A szemaforműveletek „oszthatatlansága" és az egyidejűség áll szemben egymással. Az alábbi példával tehetjük könnyen elképzelhetővé a problémát: tegyük föl, hogy a kritikus szakaszban éppen nem tartózkodik folyamat, amikor egyszerre két folyamat érkezik oda. Hogy lehet az, hogy a kettő közül csak az egyiket engedi tovább, a másikat meg nem; az egyiknek foglaltat, a másiknak szabadot mutat, jól lehet ugyanakkor még semmilyen folyamat sincs a védett szakaszban?

Bemutatunk egy mechanizmust arra, hogyan oldható föl a fenti konfliktushelyzet, miközben megadjuk a szemafortípus „finomabb" szerkezetét. A szemafor P-műveletének működése három –egymásutáni– „mikrolépésben" zajlik.

1. Minden időpillantban érkezhetnek folyamatoktól igények, amelyek egy *flag*-et beállítanak egy meghatározott értékre. A flag lehet pl. *IttVagyok*, ill. *NemVagyokItt* értékkel definiált. Minden egyes folyamathoz (szemaforonként) önálló flag tartozik. Ezért ezek egyidejű állítása egymást nem zavarhatják. A szemafor-flag-et állító folyamat(ok) az állítás után inaktiválódnak automatikusan.

```
Típus Flag = (IttVagyok, NemVagyokItt)
      Flagek = Tömb(1..N: Flag)              {N: a rendszerbeli folyamatok össz darabszáma}
```

2. A szemafor szabad (azaz a kritikus szakasz üres) állapotban *adott időegység*enként egy IttVagyok állapotú folyamatot beléptet a kritikus szakaszba és a folyamatflag-et visszájárafordítja, majd sajátmaga is foglalt állapotba kerül.

3. Előkészít a következő időegységben érkező igények fogadására egy újabb *flag-vektor*t (azaz minden folyamat ebben a pillanatban NemVagyokItt értékű flag-gel rendelkezik). Ezzel éri el, hogy az időben korábban érkezők, korábban juthassanak aktív állapotba. Ha persze az előző időegységben nem futott be igény, akkor újabb vektorra nincs szükség. Vegyük észre, hogy mivel összesen N folyamat fut a rendszerben (s minden folyamat csak legfeljebb egyszer képes flag-ét IttVagyok-ra

állítani, mivel utána inaktiválódik), ezért „legfeljebb” N db flag-vektort kell fenntartanunk az egyes szemaforokhoz.

```
Típus VárakozásFlagek = Tömb(1..N: Flagek)
```

Megjegyzések:

1. Úgy is értelmezhetjük a fent leírtakat, hogy az első lépésben az érdekelt folyamatok ideiglenesen beállnak egy speciális sorba, majd a 2. lépésben közülük egy –ha lehet– továbbjut a kritikus szakaszba.

2. Látszik, hogy a **VárakozásFlagek** egy olyan új típust (valójában nem is az „első felindultságunkban” írt *tömb*félét) jelent, amelyre vonatkozólag értelmesek a következő műveletek:

   Sorból, ti. ha kiürült a legkorábbi időegységhez tartozó Flagek-szegemens,

   Sorba, ti. amikor a következő időegységben csupa NemVagyokItt komponens

   ⇒ **Sor(Flagek)**

3. Látszik, hogy a **Flagek** is egy az előzetes terveinktől eltérő típuskonstrukcióval lehetne kifejezően jellemezni, amelyre értelmezi az alábbi műveletet kellene:

   Sorból, ti. egy IttVagyok flag-ű folyamat sorból kivételekor.

   ⇒ **Sor(Flag)**

   PrSorba, ti. amikor az aktív Flag-szegmens folyamathoz tartozó flag-ét IttVagyok-ra állítjuk (valamilyen –pl. sorszám– jellemzője alapján „rendezve” a folyamatokat).

   ⇒ **PrioritásiSor(Flag)**

   Mindezt az alábbi típusdefinícióval foglalhatjuk össze[6]:

```
Típus Szamafor = Rekord
                    ( állás: SzemaforÁllás
                      vSor: VárakozásiFlagek )

     SzemaforÁllás = (Szabad, Foglalt)
     VárakozásFlagek = Sor(Flagek)
     Flagek = PrioritásiSor(Flag)

Eljárás Sorból(Változó vS:VárakozásFlagek, szegmens:Flagek)
   {ef: nem Üres(vS)
    uf: Üres(szegmens) ... itt a „sorszerű” működést nem formalizáltuk ... }
Eljárás Sorba(Változó vS:VárakozásFlagek, Konstans szegmens:Flagek)
   {ef: Üres(szegmens)
    uf: ... a „sorszerű” működést nem formalizáltuk ... }
Eljárás PrSorból(Változó fS:Flagek, f:Flag)
   {ef: nem Üres(fS)
    uf: fS=IttVagyok ... a „prioritási sorszerű” működést nem formalizáltuk ... }
Eljárás PrSorba(Változó fS:Flagek, Konstans i:1..N, f:Flag)
   {ef: f=IttVagyok és fS_i=NemVagyokItt
    uf: ... a „prioritási sorszerű” működést nem formalizáltuk ... }
```

4. Látszik, hogy a „párhuzamosság szekvencialitássá” alakításának egyik alapötlete az, hogy feltételezi, minden folyamatnak egyedi azonosítója (indexe) van, amit prioritást meghatározó jellemzőként használ föl.

5. A választott időegység finomításával a tényleges időbeliséget megközelítően lehet követni. Ennek persze korlátja a 2. és 3. lépés végrehajtási idejének hossza.

6. Vannak a helyfoglalást csökkentő megfontolások, amikre most nem térünk ki.

---

[6] A jelölések tekintetében igazodtunk a EST-ben alkalmazottakhoz.

Hivatkozások

MfSSP: G.R. Endrews – A Method for Solving Syncronization Problems. Science of Computer Programming 13 (1989/90), North-Holland, 1990

PA&Sz: Varga László – Programok analízise és szintézise. Akadémiai Kiadó, 1981

EST: Szlávi Péter – Előadás a sorozattípusokról (*µlógia Szilánkok 7*). ELTE Ált.Szám.tud. Tsz., 1993

PB: Szlávi Péter, Zsakó László – Módszeres programozás: Programozási bevezető (*µlógia 17*). ELTE Ált.Szám.tud. Tsz., 1993

Index