MECANISM DE EXCLUDERE MUTUALĂ ŞI SINCRONIZARE BAZAT PE SEMAFOARE

Nicolae Robu

Universitatea Tehnică din Timişoara

Rezumat Lucrarea prezintă soluţiile adoptate în implementarea semafoarelor şi a funcţiilor conexe lor, în cadrul executivului de timp real **RTC86**, conceput şi realizat de autor, sub sistemul de operare **MS-DOS**, ca extensie a mediului de programare **TURBO C**.

Se evocă noţiunea de semafor, iar apoi se arată modul de implementare a semafoarelor propriu-zise, în executivul **RTC86**.

Se pun în evidenţă funcţiile de gestiune a semafoarelor în cadrul executivului **RTC86** şi principiile care au stat la baza implementării lor. Textul C al acestor funcţii este, şi el, redat.

Cuvinte cheie: excludere mutual, sincronizare, semafor, **FIFO**, scheduler, "time-out".

1. Introducere

După cum se cunoaşte, semafoarele, introduse în 1965, de către olandezul **E.W.Dijkstra** [3], joacă un rol de primă importanţă în rezolvarea problemelor de excludere mutuală şi de sincronizare în programarea concurent.

Un semafor **S** este un ansamblu format dintr-o variabilă întreagă şi o coadă de aşteptare. În momentul creerii semaforului, variabilei şi se atribuie o valoare iniţială ne negativ, iar coada este vid [2]. Întrucât variabila este supusă doar la operaţii de incrementare şi decrementare, ea poartă denumirea de contor. Coada semafoarelor este gestionată, de obicei, fie după principiul *FIFO* (*First In, First Out*), fie pe bază de priorităţi. În executivul de timp real **RTC86**, conceput şi implementat de autor, s-a adoptat gestiunea FIFO.

Se precizează că **RTC86** este un executiv grefat pe sistemul de operare **MS-DOS**, prezentându-se ca o extensie a mediului de programare **TURBO C**. Cu ajutorul acestei extensii, mediului **TURBO C** şi se conferă facilităţi de dezvoltare a programelor multitasking în timp real. Prin capabilităţile de care dispune, prin eficienţa de utilizare a procesorului şi timpii de răspuns pe care le asigură la nivelul programelor de aplicaţie, executivul **RTC86** se înscrie în rândul instrumentelor software destinate domeniului informaticii industriale [3,5].

Se prezintă, în continuare, soluţiile adoptate pentru implementarea semafoarelor în executivul de timp real **RTC86**.

2. Tipul de dată SEMAPHORE

În implementarea mecanismelor bazate pe semafoare în cadrul unui executiv de timp real, o primă decizie trebuie luată în ceea ce priveşte statutul pe care îl are semaforul propriu-zis. În executivul **RTC86**, s-a adoptat ca semaforul să fie o structură, cu numele **SEMAPHORE**, cuprinzând:

- un tablou de tipul *unsigned short* (prescurtat: usshort), cu *MAX_TSK* elemente, dedicat a servi drept coadă a semaforului; s-a dat numele coadă acestui tablou;
- o variabilă de tipul *short*, reprezentând contorul semaforului; s-a dat numele *contor* acestei variabile;
- o variabilă de tipul *pointer* către un întreg scurt fără semn, având rolul de a asista intrarea în coada semaforului; s-a dat numele *pi* acestei variabile, ca abreviere de la *pointer de intrare*;
- o variabilă de tipul *pointer* ctre un întreg scurt fără semn, având rolul de a asista ieşirea din coada semaforului; s-a dat numele pe acestei variabile, ca abreviere de la *pointer de ieire*.

Tipul de dat *SEMAPHORE* se introduce, deci, aşa cum se arată în figura 1.

```
typedef struct {
        usshort coada[MAX_TSK];
        short contor;
        usshort *pi;
        usshort *pe;
      } SEMAPHORE;
```

Notă: usshort este prescurtare de la unsigned short.

Figura 1. Tipul de dată *SEMAPHORE*.

Este firesc ca toate semafoarele de care dispune un sistem să fie grupate sub forma unui tablou. Dacă admitem că numărul maxim de semafoare este *MAX_SEM*, atunci acest tablou, din motive lesne de înţeles, de clasă *extern* -fie *_sem* numele lui-, va fi declarat astfel:

```
extern SEMAPHORE _sem[MAX_SEM];
```

Figura 2. Declararea tabloului de semafoare *_sem* [].

În aceast situaţie, un semafor concret va fi exploatat cu ajutorul indicelui său în cadrul tabloului de semafoare, introducând un indentificator sugestiv pentru fiecare indice.

3. Funcţiile de gestiune a semafoarelor

Dată fiind importanţa şi delicateţea semafoarelor într-un sistem cu prelucrare concurentă, se impune ca utilizatorul să poată face referiri la ele doar prin intermediul unor funcţii ale executivului. Clasic [1, 5], acestea sunt cele prezentate, sintetic, în figura 3.

```
                    void s_init (void)
/*   iniţializează tabloul de semafoare prin      */
/*   atribuirea valorii NULL componentelor        */
/*        pi ale elementelor sale                 */

                    usshort s_creat (short valinit)
/*   alocă un element al tabloului _sem [],       */
/*   atribuie componentei contor a elementului    */
/*   alocat valoarea valinit şi furnizează        */
/*        indicele acestui element                */

                    void s_destroy (usshort ind_sem)
/*   dezalocă elementul tabloului _sem [] cu      */
/*   indicele ind_sem, punîndu-l la dispoziţia    */
/*        funcţiei s_creat ()                     */

usshort s_wait (usshort ind_sem, usshort timeout)
/*   efectuează funcţia P asupra semaforului      */
/*   cu indicele ind_sem şi înregistrează         */
/*   valoarea argumentului timeout ca timp        */
/*   limită în care procesul curent trebuie să    */
/*   treacă de semafor; returnează o valoare      */
/*   nulă dacă trecerea se produce înaintea       */
/*   expirării timpului, altfel -una nenulă       */

                    void s_signal (usshort ind_sem)
/*   efectuează funcţia V asupra semaforului      */
/*        cu indicele ind_sem                     */
```

Figura 3. Funcţiile relative la semafoare.

Funcţia *s_init ()*

Această funcţie are rolul de a iniţializa tabloul de semafoare, prin stabilirea valorii *NULL* pentru componetele *pi* ale tuturor elementelor sale.

Textul funcţiei *s_init ()* este redat în figura 4.

```
1   void s_init (void)
2    {
3   register SEMAPHORE *s;
4   register unsigned j;
5   s=&_sem[0];
6   for (j=0;j<MAX_SEM;j++) {
7         s->pi=NULL;
8         s++;
9    }
10 }
```

Figura 4. Textul funcţiei *s_init ()*.

Funcţia *s_creat ()*

Din punct de vedere al utilizatorului, funcţia *s_creat ()* are rolul de a identifica un element liber al tabloului de semafoare *_sem* [] şi de a-l aloca, atribuind componentei sale *contor* valoarea precizat prin argumentul *valinit*; indicele elementului este făcut cunoscut apelantului prin returnare. În plus, acestei funcţii îi revine şi sarcina de a pregăti semaforul alocat pentru acţiunile pe care funcţiile *s_wait ()* şi *s_signal ()* le vor intreprinde asupra sa. Achitarea acestei sarcini comportă poziţionarea pointerilor *pi* şi *pe* din componenţa tipului de dată **SEMAPHORE**, către primul element al cozii.

Rezultă pentru funcia *s_creat ()* textul din figura 5.

```
1   usshort s_creat (short valinit)
2   {
3    register SEMAPHORE *s;
4    register unsigned n;
5    s=&_sem[0];
6    n=0;
7    _lock_();
8    while ((s->pi)&&(n<MAX_SEM)) {
9      s++;
10     n++;
11   }
12   /* în ciclul while, s-a considerat că un        */
13   /* semafor încă nealocat se distinge prin       */
14   /* valoarea NULL a componentei sale pi          */
15   /* (a se vedea s_init () şi s_destroy ())       */
16   if (n<MAX_SEM) {
17     s->contor=valinit;
18     s->pi=&(s-coada[0]);
19     s->pe=&(s-coada[0]);
20    }
21   else {
22      /* eşec: nici un semafor    */
23      /* nu a fost găsit liber    */
24   }
25   _unlock_();
26   return (n);
27   /* se furnizează indicele semaforului alocat */
28 }
```

Notă: *_lock_()* dezactivează întreruperile, iar *_unlock_()* le activeaz.

Figura 5. Textul funcţiei *s_creat ()*.

Funcţia *s_destroy ()*

Din punct de vedere al utilizatorului, funcţia *s_destroy ()* are rolul de a dezaloca elementul tabloului de semafoare *_sem* [] cu indicele specificat prin valoarea argumentului ei. Dezalocarea se realizează prin readucerea componentei *pi* a acestui element la valoarea iniţială *NULL*. În urma acţiunii funcţiei *s_destroy ()*, elementul în cauză este lăsat la dispoziţia funcţiei *s_creat ()*, în vederea unei noi alocări.

Se precizează că regulile care guvernează lucrul cu semafoare **interzic** distrugerea unui semafor a cărui coadă este nevidă. Fidel fiind acestei reguli, funcţia *s_destroy ()* va efectua o verificare a cozii semaforului asupra căruia planează perspectiva distrugerii şi îl va distruge doar dacă coada sa este vid. Altfel, acţiunea funcţiei se va rezuma la generarea unui mesaj de eroare.

Textul funcţiei *s_destroy ()* este dat în figura 6.

```
1   void s_destroy (usshort ind_sem)
2  {
3   register SEMAPHORE *s;
4   s=&_sem[ind_sem];
5   _lock_();
6   if (s->contor<0) {
7     /* eroare: se încearcă distrugerea unui    */
8     /* semafor a cărui coadă este nevidă       */
9    }
10  else {
11  s->pi=NULL;
12    /* se atribuie valoarea NULL componentei  */
13    /* pi a semaforului care se distruge      */
14  }
15   _unlock_();
16 }
```

Figura 6. Textul funcţiei *s_destroy ()*.

Funcţia *s_wait ()*

Funciţa *s_wait ()* are rolul de a materializa operaţiile care definesc primitiva **P**, cu referire la semaforul precizat prin primul său argument. O parte dintre aceste operaţii, şi anume cele de scoatere a procesului curent dintre procesele rulabile şi de blocare a acestui proces, sunt asigurate prin apelul unei funcţii numită *sleep ()*.

Cel de-al doilea argument al funciei *s_wait ()* este destinat transmiterii lui la funcţia *sleep ()*, pentru ca aceasta să asigure limitarea timpului cât procesul în cauză poate rămâne blocat la semafor. Ajungerea unui proces pus în aşteptare la un semafor în situaţia deblocării automate reprezintă o anomalie care trebuie tratată în mod corespunztor de către utilizator. În sprijinul acestuia, funcţia *s_wait ()* returnează o valoare *unsigned short*, nenulă, în cazul în care anomalia s-a petrecut, şi nu, altfel. Evident, funcţia *scheduler ()* este cea care generează primar această valoare; funcţia *s_wait ()* o preia de la *scheduler ()* prin intermediul funcţiei *sleep ()*.

Legat de operaţia de înscriere a taskului curent în coada de aşteptare la semafor, se precizează că ea trebuie să se efectueze circular, prin repoziţionarea pointerului *pi* pe primul element al tabloului care implementează coada, după ce el a servit înscrierii unui task în ultimul element al acestui tablou.

Conform celor de mai su:, funcţia *s_wait ()* se poate implementa prin textul C redat în figura 7.

```
1   usshort s_wait (usshort ind_sem, usshort timeout)
2  {
3   register SEMAPHORE *s;
4   register unsigned aux;
5   s=&_sem[ind_sem];
6   aux=0;
7   /* se inițializează variabila aux pentru a    */
8   /* indica ieirea normal din funcia s_wait     */
9   _lock_();
10  if (--s->contor) {
11   *s->pi=_task_crt;
12    /* se înscrie procesul curent în             */
13    /* coada de aşteptare la semafor             */
14  if (s->pi==&(s->coada[MAX_TSK-1])) {
15    s->pi=&(s->coada[0]);
16    /* se actualizează valoarea pointerului */
17    /* de intrare în coadă, astfel încît să      */
18    /* indice primul element al tabloului        */
19   }
20  else {
21    s->pi++;
22    /* se actualizează valoarea pointerului      */
23    /* de intrare în coadă, astfel încît să      */
24    /* indice următorul element al tabloului     */
25   }
26  aux=sleep (timeout);
27  /* se elimină procesul curent dintre          */
28  /* procesele rulabile şi se autoblochează;    */
29  /* se returnează o valoare nenulă în caz      */
30  /*     de "timeout" şi nulă altfel            */
31  }
32   _unlock_();
33  return (aux);
34 }
```

Figura 7. Textul funcţiei *s_wait ()*.

Este lesne de înţeles că, dacă în urma decrementrii contorului semaforului, valoarea acestuia este negativă, funcţia *s_wait ()* se va derula în două reprize. Prima repriz va cuprinde liniile [1...26], iar a doua repriză -liniile [26...34]. Asta înseamnă că o parte a funcţiei *sleep ()* -linia **26**- se va rula în repriza întâi, iar cealaltă parte -în repriza a doua. Între cele două reprize, va avea loc rularea altor taskuri, printre care, în mod normal, trebuie să fie şi cele care au executat înaintea taskului curent funcia *s_wait ()* cu referire la acelaşi semafor. Pentru a atrage atenţia asupra acestor aspecte a fost supraimprimată linia **26** din figura 7.

Funcţia s_signal ()

Funcţia *s_signal ()* are rolul de a materializa operaţiile care definesc primitiva **V**, cu referire la semaforul precizat prin argumentul său. O parte dintre aceste operaţii, şi anume cele de deblocare a procesului aflat în capul cozii semaforului şi de înscriere a acestui proces în rândul proceselor rulabile, sunt asigurate prin apelul unei funcţii, denumit *wakeup ()*.

Legat de operaţia de determinare a procesului aflat în capul cozii semaforului şi de eliminare a lui din această coadă, se precizează că ea trebuie să se efectueze circular, prin poziţionarea pointerului pe către primul element al tabloului care implementează coada, după ce el a servit extragerii unui proces înscris în ultimul element al acestui tablou.

Rezultă următorul text pentru funcţia *s_signal ()*:

```
1   void s_signal (usshort ind_sem)
2   {
3   register SEMAPHORE *s;
4   register usshort tsk;
5   sem=&_sem [ind_sem];
6    lock_();
7   if (++s->contor<=0) {
8      tsk=*s-pe;
9      /* se determină procesul aflat      */
10     /* în capul cozii de aşteptare      */
11     if (s->pe==&(s->coada[MAX_TSK-1])) {
12        s->pe=&(s->coada[0]);
13        /* se actualizează valoarea pointerului */
14        /* de ieşire din coadă, astfel încît să  */
15        /* indice primul element al tabloului  */
16     }
17     else {
18        s->pe++;
19        /* se actualizează valoarea pointerului */
20        /* de ieşire din coadă, astfel încît să  */
21        /* indice următorul element al tabloului */
22     }
23  wakeup (tsk);
24  /* se deblochează procesul găsit în capul    */
25  /* cozii de aşteptare la semafor şi se       */
26  /* înscrie printre procesele rulabile        */
27  }
28   _unlock_();
29  }
```

Figura 8. Textul funcţiei *s_signal ()*.

Se face observaţia că apelul funcţiei *wakeup ()* va provoca o acţiune de comutare, prin care este posibil ca procesul curent să piardă controlul asupra procesorului, chiar dacă el rămâne în continuare rulabil, în favoarea procesului proaspt deblocat cu ajutorul acestei funcţii. Rezultă, prin urmare, o posibilă fragmentare a rulării funcţiei *s_signal ()*, în două reprize, în cazul în care, în urma incrementării contorului semaforului, valoarea sa este negativ sau nulă, indicând că cel puţin un proces se află în coada de aşteptare la semafor. Prima repriză va cuprinde liniile [1...23], iar a doua repriză -liniile [23...29]. Aceasta înseamn că o parte a funcţiei wakeup () -linia **23**- se va rula în repriza întâi, iar cealaltă parte -în repriza a doua. Între cele două reprize, poate avea loc rularea altor procese. Pentru a atrage atenţia asupra acestor aspecte a fost supraimprimat linia **23** din figura 8.

4. Concluzii

Soluţiile prezentate sunt înglobate în executivul de timp real **RTC86**. Ele conferă mecanismelor de excludere mutuală şi de sincronizare ale acestui executiv o utilizabilitate comodă şi o viteză de execuţie eficientă.

Bibliografie

1. **ALLWORTH, S.T.** -Introduction to Real-Time Software Design, Macmillan Press Ltd., London, 1981.
2. **BALTAC V. ş.a.** -Sisteme interactive şi limbaje conversaţionale. Utilizare şi proiectare, Editura Tehnică, Bucureşti, 1984.
3. **DIJKSTRA, E.W.** -Solution of a Problem in Concurrent Programming Control, Communications ACM, Nr. 8, 1965, p. 569.
4. **PEREZ, J.P.** -Systemes Temps Reel -Methodes de Specification et de Conception, Dunod, Paris, 1990.
5. **TSCHIRHART, D.** -Commande en Temps Reel, Dunod, Paris, 1990.