MECANISM DE EXCLUDERE MUTUALĂ PRIN BLOCURI RESURSĂ

Nicolae Robu

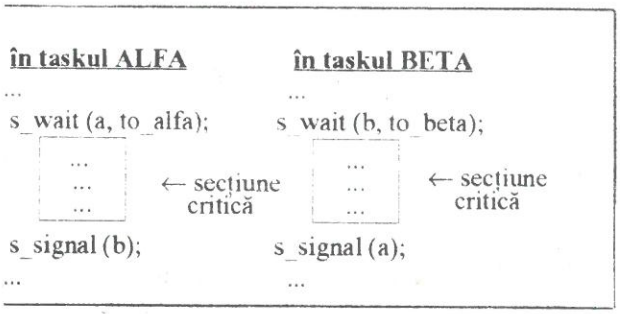
Universitatea Tehnică din Timișoara

Rezumat: Lucrarea prezintă soluțiile adoptate în implementarea unui mecanism de excludere mutuală prin blocuri resursă, în cadrul executivului de timp real **RTC86**, conceput și realizat de autor, sub sistemul de operare MS-DOS, ca extensie a mediului de programare TURBO C.
Se evocă noțiunea de bloc resursă, iar apoi se arată modul de implementare a blocurilor resursă propriu-zise, în executivul **RTC86**.
Se pun în evidență funcțiile mecanismului de excludere mutuală prin blocuri resursă, înglobat în executivul **RTC86**, și principiile care au stat la baza implementării lor. Textul C al acestor funcții este, și el, redat.

Cuvinte cheie: excludere mutuală, semafor, bloc resursă, gestionar de resursă, scheduler, "time-out".

1. Introducere

Este cunoscut, un neajuns al semafoarelor, aparent minor, dar, în practică sursă a numeroase erori, greu de depistat în aplicațiile complexe, este că, la nivelul execuției programelor, nu există garanția delimitării corecte a secțiunilor critice prin funcțiile ce implementează primitivele **P**, respectiv **V**.
De exemplu, o dublă eroare de programare ca cea surprinsă în figura 1, nu va putea fi semnalată la compilare, dar nici în timpul rulării, conducînd la un comportament al programului greu de explicat.



Notă: *s_wait ()* și *s_signal ()* sînt funcțiile de gestionare a semafoarelor, corespunzătoare primitivelor P, respectiv V.

Figura 1. Exemplu de utilizare greșită a primitivelor P și V.

Pentru a se putea semnala, în timpul rulării, existența unor situații ca cea din figura 1, soluția este să se extindă accepțiunea consacrată a noțiunii de semafor [2], cu o componentă în care să se înregistreze indexul procesului ce, executînd primitiva P asupra semaforului, reușește să-și inițieze secțiunea critică. O asemenea extindere transformă semaforul într-un bloc resursă [4]. Componenta specifică a blocului resursă oferă primitivei V posibilitatea de a o consulta și folosi, pentru condiționarea execuției sale propriu-zise de concordanța dintre indexul procesului în care această primitivă este programată și valoarea pe care ea - componenta - o conține înregistrată.
Se amintește că se obișnuiește ca procesul al cărui index este înregistrat în componenta specifică a unui bloc resursă să fie considerat *proprietarul* curent al *resursei* în cauză la un moment, și că pentru perechea de primitive P și V aferente unui bloc resursă este folosit termenul *"gestionar de resursă"*.
Se prezintă, în continuare, soluțiile adoptate pentru implementarea blocurilor resursă și a funcțiilor conexe, în executivul de timp real **RTC86**, conceput și realizat de autor.
Se precizează că **RTC86** este un executiv grefat pe sistemul de operare **MS-DOS**, prezentîndu-se ca o extensie a mediului de programare **TURBO C**. Cu ajutorul acestei extensii, mediului **TURBO C** i se conferă facilități de dezvoltare a programelor multitasking în timp real. Prin capabilitățile de care dispune, prin eficiența de utilizare a procesorului și timpii de răspuns pe care le asigură la nivelul programelor de aplicație, executivul **RTC86** se înscrie în rîndul instrumentelor software destinate domeniului informaticii industriale [3,4].

2. Tipul de dată RESOURCE

În implementarea, în cadrul unui executiv de timp real, a unui mecanism de excludere mutuală cu ajutorul blocurilor resursă, o primă decizie trebuie luată în ceea ce privește statutul pe care îl are blocul resursă propriu-zis. În executivul **RTC86**, s-a adoptat ca blocul resursă să fie o structură, cu numele **RESOURCE**, cuprinzînd:

- un tablou de tipul *unsigned short*, cu MAX_TSK (numărul maxim al proceselor acceptabile în sistem) elemente, dedicat a servi drept coadă de așteptare la blocul resursă; s-a ales numele coadă pentru acest tablou;
- o variabilă de tipul *short*, reprezentînd contorul blocului resursă; s-a ales numele *contor* pentru această variabilă;
- o variabilă de tipul *pointer* către un întreg scurt fără semn, avînd rolul de a asista **intrarea** în coada blocului resursă; s-a ales numele *pi* pentru această variabilă;
- o variabilă de tipul *pointer* către un întreg scurt fără semn, avînd rolul de a asista **ieșirea** din coada blocului resursă; s-a ales numele *pe* pentru această variabilă;
- o variabilă de tipul *unsigned short*, dedicată înregistrării indexului proprietarului curent al resursei căreia îi corespunde blocul resursă; s-a ales numele *prop* pentru această variabilă.

Tipul de dată *RESOURCE* se va introduce, deci, așa cum se arată în figura 2.

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

Notă: *usshort* este prescurtare pentru *unsigned short*.

Figura 2. Tipul de dată *RESOURCE*

Este firesc ca toate blocurile resursă disponibile într-un sistem să fie grupate sub forma unui tablou. Dăm numele *_res* [] acestui tablou. Dacă admitem că numărul maxim de blocuri resursă este **MAX_RES**, atunci tabloul *_res* [], care, din motive lesne de înțeles, trebuie să fie de *clasă extern*, va fi declarat așa cum se arată în figura 3.

```
extern RESOURCE  _res [MAX_RES];
```

Figura 3. Declararea tabloului de blocuri *resursă_res* []

Pe baza celor de mai sus, un bloc resursă concret va fi exploatat cu ajutorul indicelui său în cadrul tabloului de blocuri resursă, introducînd cîte un identificator sugestiv, pentru fiecare indice.

3. Funcțiile de gestiune a blocurilor resursă

Dată fiind importanța și delicatețea blocurilor resursă într-un sistem cu prelucrare concurentă, se impune ca utilizatorul să poată face referiri la ele doar prin intermediul unor funcții ale executivului. Clasic [1, 4], acestea sunt cele prezentate, sintetic, în figura 4.

```
            void r_init (void)
/*   inițializează tabloul de blocuri                  */
/*   resursă, prin atribuirea valorii  NULL            */
/*   componentelor pi  ale elementelor sale            */

            usshort r_creat (void)
/*   alocă un element al tabloului _res [],            */
/*   atribuie componentei  contor a elemen-            */
/*   tului alocat valoarea 1 și furnizează             */
/*   indicele acestui element                          */

            void r_destroy (usshort ind_res)
/* dezalocă elementul tabloului _res []                */
 /* cu indicele ind_res, repunîndu-l la                */
 /*     dispoziția funcției r_creat ()                 */

usshort r_wait (usshort ind_res, usshort timeout)
 /* efectuează operația P asupra blocului              */
 /* resursă cu indicele ind_res și reține              */
 /* valoarea argumentului timeout ca timp              */
 /* limită în care procesul curent poate               */
 /* rămîne  blocat la blocul  resursă;                 */
 /* returnează o valoare nulă, dacă deblo-             */
 /* carea se produce înaintea expirării                */
 /* timpului, altfel -una nenulă; înregis-             */
 /* trează în cîmpul prop al blocului                  */
 /* resursă ind_res  valoarea indexului                */
 /* procesului curent, cînd acesta este                */
 /*    autorizat să-și continue rularea                */

            void r_signal (usshort ind_res)
 /* efectuează operația V  asupra blocului             */
 /* resursă cu indicele ind_res, dacă                  */
 /* proprietarul  resursei corespondente               */
 /* acestuia  este procesul curent, altfel             */
 /*          semnalează eroare                         */
```

Figura 4. Funcțiile de exploatare a blocurilor resurs

Funcția *r_init ()*

Această funcție are rolul de a inițializa tabloul de blocu resursă, prin stabilirea valorii *NULL* pentru componente *pi* ale tuturor elementelor sale.
Textul funcției *r_init ()* este redat în figura 5.

```
1  void r_init (void)
2  {
3  register RESOURCE *r;
4  register unsigned j;
5  r=&_res[0];
6  for (j=0;j<MAX_RES;j++) {
7    r→pi=NULL;
8    r++;
9  }

10 }
```

Figura 5. Textul funcției *r_init ()*

Funcția *r_creat ()*

Din punct de vedere al utilizatorului, funcția *r_creat ()* a rolul de a identifica un element liber al tabloului de blocu resursă *_res* [] și de a-l aloca, atribuind componentei sa *contor* valoarea **1**; indicele elementului este făcut cunosc apelantului prin returnare. În plus, acestei funcții îi revi și sarcina de a pregăti blocul resursă alocat pentru acțiuni pe care funcțiile *r_wait ()* și *r_signal ()* le vor intreprin asupra sa. Achitarea acestei sarcini comportă poziționar pointerilor *pi* și *pe* din componența tipului de da ***RESOURCE***, către primul element al cozii.
Rezultă, pentru funcția *r_creat ()*, textul din figura 6.

```
1  usshort r_creat (void)
2  {
3   register RESOURCE *r;
4   register unsigned n;
```

```
5   r=&_res[0];
6   n=0;
7   _lock_();
8   while ((r→pi)&&(n < MAX_RES)) {
9    r++;
10   n++;
11  }
12  /* în ciclul while, s-a considerat că un bloc      */
13  /* resursă încă nealocat se distinge prin          */
14  /* valoarea NULL a componentei sale pi             */
15  /*  (a se vedea r_init () și r_destroy ())         */
16  if (n < MAX_RES) {
17   r → contor=1;
18   r →pi=&(r → coada[0]);
19   r →pe=&(r → coada[0]);
20  }
21  else {
22   /* eșec: nici un bloc resursă                     */
23   /* nu a fost găsit liber                          */
24  }
25  _unlock_();
26  return (n);
27  /* se furnizează indicele                          */
28  /* blocului resursă alocat                         */
29  }
```

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

Figura 6. Textul funcției *r_creat()*

Funcția *r_destroy()*

Din punct de vedere al utilizatorului, funcția *r_destroy* () are rolul de a dezaloca elementul tabloului de blocuri resursă *_res* [] cu indicele specificat prin valoarea argumentului ei. Dezalocarea se realizează prin readucerea componentei *pi* a acestui element la valoarea inițială *NULL*. În urma acțiunii funcției *r_destroy ()*, elementul în cauză este lăsat la dispoziția funcției *r_creat ()*, în vederea unei noi alocări.

Se precizează că regulile care guvernează lucrul cu blocuri resursă **interzic** distrugerea unui bloc a cărui coadă este nevidă. Fidelă fiind acestei reguli, funcția *r_destroy ()* va efectua o verificare a cozii blocului resursă asupra căruia planează perspectiva distrugerii și îl va distruge doar dacă coada sa este vidă. Altfel, acțiunea funcției se va rezuma la generarea unui mesaj de eroare.

Textul funcției *r_destroy ()* este redat în figura 7.

```
1   void r_destroy (usshort ind_res)
2   {
3    register RESOURCE *s;
4    r=&_res[ind_res];
5    _lock_();
6    if (r → contor < 0) {
7    /* eroare: se încearcă distrugerea unui bloc      */
8    /* resursă a cărui coadă este nevidă              */
9   }
10   else {
11   r → pi=NULL;
12   /* se atribuie valoarea NULL componentei          */
13   /* pi a blocului resursă care se distruge         */
14   }
15   _unlock_();
16  }
```

Figura 7. Textul funcției *r_destroy()*

Funcția *r_wait()*

Funcția *r_wait ()* are rolul de a materializa operațiile care definesc primitiva **P**, cu referire la blocul resursă precizat prin primul său argument. O parte dintre aceste operații, și anume, cele de scoatere a procesului curent dintre procesele rulabile și de blocare a acestui proces, sunt asigurate prin apelul unei funcții numită *sleep ()*.

Cel de-al doilea argument al funcției *r_wait ()* este destinat transmiterii lui la funcția *sleep ()*, pentru ca aceasta să asigure limitarea timpului cît procesul în cauză poate rămîne blocat la blocul resursă. Ajungerea unui proces pus în așteptare la un bloc resursă în situația deblocării automate reprezintă o anomalie care trebuie tratată în mod corespunzător de către utilizator. În sprijinul acestuia, funcția *r_wait ()* returnează o valoare *unsigned short*, nenulă, în cazul în care anomalia s-a petrecut, și nulă, altfel. Evident, funcția *scheduler ()* este cea care generează primar această valoare; funcția *r_wait ()* o preia de la *scheduler ()* prin intermediul funcției *sleep ()*.

Legat de operația de înscriere a procesului curent în coada de așteptare la blocul resursă, se precizează că ea trebuie să se efectueze circular, prin repoziționarea pointerului *pi* pe primul element al tabloului care implementează coada, după ce el a servit înscrierii unui proces în ultimul element al acestui tablou.

Conform celor de mai sus, funcția *r_wait ()* se poate implementa prin textul C redat în figura 8.

```
1   usshort r_wait (usshort ind_res, usshort timeout)
2  {
3   register RESOURCE *r;
4   register usshort aux;
5   r=&_res[ind_res];
6   aux=0;
7   /* se inițializează variabila aux pentru a        */
8   /* indica ieșirea normală din funcția r_wait      */
9   _lock_();
10  if (--r → contor < 0) {
11   *r → pi=_task_crt;
12   /* se înscrie procesul curent în                  */
13   /* coada de așteptare la resursă                  */
14   if (r → pi= =&(r → coada[MAX_TSK-1])) {
15    r → pi=&(r → 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    r → 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  r → prop= _task_crt;
33  _unlock_();
34  return (aux);
35 }
```

Figura 8. Textul funcţiei *r_wait ()*.

Este lesne de înţeles că, dacă în urma decrementării contorului blocului resursă valoarea acestuia este negativă, funcţia *r_wait ()* se va derula în două reprize. Prima repriză va cuprinde liniile [1...26], iar a doua repriză -liniile [26...35]. Asta înseamnă că o parte a funcţiei *sleep ()* -linia **26**- se va rula în repriza întîi, iar cealaltă parte - în repriza a doua. Între cele două reprize, va avea loc rularea altor procese, printre care, în mod normal, trebuie să fie şi cele care au executat înaintea procesului curent funcţia *r_wait ()* cu referire la acelaşi bloc resursă. Pentru a atrage atenţia asupra acestor aspecte a fost supraimprimată linia **26** din figura 8.

Funcţia *r_signal ()*

Funcţia *r_signal ()* trebuie să debuteze cu consultarea cîmpului *prop* al blocului resursă cu indicele *ind_res*. Dacă valoarea înscrisă în acest cîmp nu coincide cu indexul procesului curent, acţiunea funcţiei *r_signal ()* se limitează la generarea unui mesaj de eroare. Altfel, funcţia *r_signal ()* efectuează, în continuare, operaţiile care definesc primitiva **V**, cu referire la blocul resursă precizat prin argumentul său. O parte dintre aceste operaţii, şi anume cele de deblocare a procesului aflat în capul cozii blocului resursă şi de înscriere a acestui proces în rîndul proceselor rulabile, sunt asigurate prin apelul unei funcţii, denumită *wakeup ()*.
Legat de operaţia de determinare a procesului aflat în capul cozii blocului resursă 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, pentru funcţia *r_signal ()*, textul din figura 9.

```
1   void r_signal (usshort ind_res)
2  {
3   register RESOURCE *r;
4   register usshort tsk;
5   r  &  res[ind_res];
6    lock ();
7   if (r → prop!    task_crt) {
8    /* eroare: procesul curent nu                */
9    /* figurează ca  proprietar */
10  }
11  else {
12   if(++r → contor <= 0) {
13     tsk=*r → pe;
14    /* se determină procesul aflat               */
```

```
15    /* în capul cozii de aşteptare               */
16    if (r →pe= =&(r → coada[MAX_TSK-1])) {
17     r → pe=&(r → coada[0]);
18     /* se actualizează valoarea pointerului     */
19     /* de ieşire din coadă, astfel încît să     */
20     /* indice primul element al tabloului       */
21    }
22    else {
23     r → pe++;
24     /* se actualizează valoarea pointerului     */
25     /* de ieşire din coadă, astfel încît să     */
26     /* indice următorul element al tabloului    */
27    }
28    wakeup (tsk);
29    /* se deblochează procesul găsit în capul    */
30    /* cozii de aşteptare la semafor şi se       */
31    /* înscrie printre procesele rulabile        */
32   }
33  }
34   unlock_();
35 }
```

Figura 9. Textul funcţiei *r_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 rămîne în continuare rulabil, în favoarea procesului proaspăt deblocat cu ajutorul acestei funcţii. Rezultă, prin urmare, o posibilă fragmentare a rulării funcţiei *r_signal ()*, în două reprize, în cazul în care, în urma incrementării contorului blocului resursă, valoarea sa este negativă sau nulă, indicînd că cel puţin un proces se află în coadă de aşteptare la resursă. Prima repriză va cuprinde liniile [1...28], iar a doua repriză - liniile [28...35]. Aceasta înseamnă că o parte a funcţiei *wakeup ()* -linia **28**- se va rula în repriza întîi, 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 supraimprimată linia **28** din figura 9.

4. Concluzii

Soluţiile prezentate sînt înglobate în executivul de timp real **RTC86**. Ele conferă acestui executiv capabilitatea de a realiza elegant şi sigur excluderea mutuală, în condiţiile unui consum redus de timp procesor.

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. **PEREZ, J.P.**: Systemes Temps Reel - Methodes de Specification et de Conception, Dunod, Paris, 1990.
4. **TSCHIRHART, D.**: Commande en Temps Reel Dunod, Paris, 1990.