Programe de aplicație

MECANISM DE EXCLUDERE MUTUALĂ PRIN FANIOANE DE EXCLUZIUNE ACTIVE

Nicolae ROBU

Universitatea Tehnică din Timișoara

Rezumat Lucrarea prezintă un mecanism original pentru asigurarea excluderii mutuale în sistemele cu prelucrare concurentă. Acest mecanism a fost integrat în executivul de timp real **RTC86**, conceput și realizat de autor, sub sistemul de operare **MS-DOS**, ca extensie a mediului de programare **TURBO C**.
Se definește noțiunea de fanion de excluziune activ, iar apoi se arată modul de implementare a fanioanelor propriu-zise, în executivul **RTC86**.
Se pun în evidență funcțiile de gestiune a fanioanelor în cadrul executivului **RTC86** și principiile care au stat la baza implementării lor. Textul **C** al acestor funcții este, și el, redat.
Cuvinte cheie: dezactivare întreruperi, fanion TAS, semafor, fanion de excluziune activ, seciune critică, interblocare.

1. Introducere

După cum este cunoscut, consacrat, excluderea mutuală se asigură fie prin dezactivarea întreruperilor, fie cu ajutorul fanioanelor *TAS* (Test And Set), fie cu ajutorul semafoarelor [1, 2, 4].

Fiecare dintre aceste metode își are avantajele și dezavantajele sale.

Metoda cea mai simplă -dezactivarea întreruperilor- are neajunsul de a implica riscul pierderii irecuperabile a unor cereri de întrerupere; în plus, această metodă nu este aplicabilă în cazul prelucrrii concurente pe sisteme multiprocesor.

Metoda bazată pe fanioane *TAS*, caracterizat, de asemenea, prin simplitate, presupune irosirea unui cuantum important din capacitatea de prelucrare a procesorului, în bucle de așteptare activă, și, în sistemele cu dispecerizare prin prioritizare, posibilitează apariția fenomenului de interblocare.

Metoda bazat pe semafoare este lipsită de dezavantajele celorlalte dou, dar, este mai complex și mai mare consumator de timp.

În lucrare se propune un mecanism original de excludere mutuală, care rezolvă problema accesului singular la resursele critice în aceeași manieră ca și mecanismele bazate pe semafoare, fiind, însă, mai simplu și, în cazul secțiunilor critice de scurtă durată, considerabil mai eficient.

2. Definirea mecanismului de excludere mutuală prin fanioane de excluziune active

Un *fanion de excluziune activ* este un ansamblu format dintr-un fanion *TAS* -pe care îl numim fanion de excluziune pasiv-, **F**, și o coadă de așteptare, **C**.

Fanionul de excluziune pasiv are rolul de a indica posibilitatea sau imposibilitatea inițierii de către un proces, la un moment, a secțiunii sale critice referitoare la o resursă pusă în corespondență cu fanionul.

Coada de așteptare servește înregistrării proceselor nesatisfăcute în tentativa lor de a-i demara secțiunea critică referitoare la resursa corespunzătoare fanionului.

Mecanismul de excludere mutuală prin fanioane de excluziune active este dezvoltat în jurul următoarei convenții:

- fanionul de excluziune este asociat resursei în cauză și numai ei;
- valoarea "**0**" a componentei **F** a fanionului arată că nici o secțiune critică referitoare la resursă nu este în derulare, iar valoarea "**1**" -contrariul;
- înainte de pătrunderea în secțiunea critică relativă la resursa în discuție, orice proces are obligația de a proceda la:
  1). efectuarea, în condiții de indivizibilitate, a operației *TAS* asupra fanionului.
  2). a). autoînscrierea în coada **C** a fanionului, *autoeliminarea din lista proceselor rulabile, autoblocarea și autoînscrierea în lista proceselor blocate, apelarea dispecerului* și repetarea, după deblocare, a obligației în curs de definire, începînd cu 1) -toate acestea dacă operația *TAS* a găsit componenta **F** la "**1**".
  b). inițierea secțiunii critice, dacă operația *TAS* a găsit componenta **F** la "**0**".
- la părăsirea secțiunii critice, orice proces are obligația de a proceda, printr-o secvență indivizibilă, la:
  1'). efectuarea operației *RES* (**RES**et) asupra componentei **F** a fanionului.
  2'). *deblocarea* tuturor proceselor înscrise în coada **C** a fanionului și, implicit, vidarea acesteia, și reînscrierea lor în lista proceselor rulabile.

3. Aspecte privind implementarea mecanismului de excludere mutuală prin fanioane de excluziune active

Mecanismul de excludere mutuală prin fanioane de excluziune active are, ca elemente de bază, două funcții, corespunzătoare, una dintre ele, operațiilor 1) și 2), iar cealaltă, operațiilor 1') și 2'); se poate deduce ușor, că, aceste funcții, vor fi plasate înaintea, respectiv la sfârșitul secțiunilor critice.

Denumim aceste funcții *_f_wait ()*, respectiv *_f_signal ()*. Vom considera că, pentru efectuarea acțiunilor scrise cu caractere italice în textele corespunzătoare operațiilor 2), respectiv 2'), funcția *_f_wait ()* face apel la o funcție pe care o denumim *sleep ()*, iar funcția *_f_signal ()* -la o funcție pe care o denumim *wakeup ()*.

Pentru a oferi posibilitatea limitării duratelor intervalelor de timp în care procesele sunt ţinute blocate, funcţia *sleep ()* dispune de un argument, având numele *timeout*, prin care se poate specifica, în perioade ale ceasului de timp real, timpul după care procesul blocat să fie automat deblocat. Dacă, la apel, valoarea argumentului *timeout* este *zero*, atunci funcţia *sleep ()* asigură o blocare pe termen nedefinit.

Pentru a se face posibilă implementarea funcţiei *sleep ()* conform celor de mai sus, este necesară prezenţa, în componenţa contextului logic al proceselor, a unei variabile cu rolul de a prelua valoarea argumentului *timeout* în scopul decrementării ei, când este nenul, la fiecare intervenţie a dispecerului cauzat de o întrerupere de la ceasul de timp real. Denumim *sltime* această variabilă.

Pe lângă *sltime*, o altă variabilă, numită *to*, este necesară în cadrul contextului proceselor, în scopul reţinerii faptului că, prin decrementarea valorii componentei *sltime*, s-a ajuns la zero, apărând, deci, situaţia de "time-out".

Se va considera, în continuare, că toate informaţiile ce ţin de contextul logic al unui proces reprezintă câmpuri ale unei structuri cu numele *CONTLOG* şi că fiecare proces are asociat un element al unui tablou de asemenea structuri, denumit *_contlog*[]. Structura *CONTLOG* avut în vedere este cea definită în figura 1.

```
typedef struct {
        unsigned int ss;
        unsigned int sp;
        unsigned char status;
        unsigned short prior;
        unsigned char to;
        unsigned int sltime;
      } CONTLOG;
```

Note: *ss* serveşte memorării conţinutului registrului *SS*
sp serveşte memorării conţinutului registrului *SP*
status serveşte memorării stării procesului
prior serveşte memorării priorităţii procesului
to serveşte memorării faptului că s-a ajuns la "time-out"
sltime serveşte memorării timpului pînă la "time-out"

Figura 1. Structura CONTLOG.

Procesele blocate de funcţia *sleep ()* vor fi înscrise într-o listă dedicată, *lista taskurilor blocate*, referită prin abrevierea *ltb*. Asupra acestei liste, se va opera cu ajutorul unor funcţii clasice, având numele: *ins_ltb ()*, *elim_ltb ()*, *urm_ltb ()* i *init_ltb ()*. Ele fac obiectul figurii 2.

```
        void ins_ltb (unsigned short ind_tsk)
/* inserează în ltb procesul cu indexul ind_tsk   */

        void elim_ltb (unsigned short ind_ltb)
/* elimină din ltb procesul cu indicele ind_tsk   */

        unsigned short urm_ltb (void)
/* determină indexul procesului aflat în capul ltb */

        void init_ltb (void)
/* iniţializează lista proceselor blocate         */
```

Figura 2. Funcţiile de gestionare a listei proceselor blocate.

Se menţionează că, pentru ca apelantul funcţiei *sleep ()* să poată afla, la revenirea din aceasta, dacă deblocarea s-a fcut normal, adică în limita de timp fixat prin argumentul *timeout*, sau forat, la expirarea timpului, funcţia returnează o valoare *unsigned int* nul, în primul caz, respectiv nenul, în al doilea; această valoare este generată, primar, de funcţia *scheduler ()*.

Ţinând seamă de cele de mai sus, funcţiile *sleep ()* şi *wakeup ()* au implementările din figurile 3, respectiv 4.

```
1   unsigned int sleep (unsigned short timeout)
2   {
3    CONTLOG *p;
4    unsigned int aux;
5    p=&_contlog[_task_crt];
6    _lock_();
7    p->status=BLOCAT;
8    p->sltime=timeout;
9    elim_lap (_task_crt);
10   ins_ltb (_task_crt);
11   aux=scheduler ();
12   _unlock_();
13   return (aux);
14  }
```

Notă: *_task_crt* este o variabilă globală ce memorează indexul procesului aflat curent în rulare

Figura 3. Textul funcţiei *sleep ()*.

```
1   void wakeup (unsigned short ind_tsk)
2   {
3    CONTLOG *p;
4    p=&_contlog[ind_tsk];
5    lock_();
6    if (p->status==BLOCAT) {
7          p->status=EXEC;
8          elim_ltb (ind_tsk);
9          ins_lap (ind_tsk);
10   }
11   _unlock_();
13  }
```

Figura . 4. Textul funcţiei *wakeup ()*.

Pentru implementarea mecanismului de excludere mutuală prin fanioane active, s-a definit un tip de dată, reprezentând fanionul propriu-zis, ca o structură cu numele *FLAG*, ce cuprinde:

- o variabilă de tipul *unsigned short*, reprezentând fanionul de excluziune pasiv; fie *f* numele acestei variabile;
- un tablou de tipul *unsigned short*, cu *MAX_TSK* elemente, dedicat a servi drept coad a fanionului de excluziune activă; fie *coada* numele acestui tablou (*MAX_TSK* este numărul maxim al proceselor acceptate în sistem).
- o variabilă de tipul *pointer* către un întreg scurt fără semn, având rolul de a asista intrarea în şi ieşirea din coada fanionului; fie *pie* numele acestei variabile.

Tipul de dată *FLAG* se introduce, deci, printr-o declaraţie de forma:

```
typedef struct {
        unsigned short f;
        unsigned short coada[MAX_TSK];
        unsigned short *pie;
    } FLAG;
```

Figura 5. Tipul de dată *FLAG*.

Este firesc ca structurile *FLAG* să fie grupate sub forma unui tablou; acesta trebuie să fie de clasă *extern*. Fie el de dimensiune *MAX_FLG* i cu numele *_flg*.

Referirile la un fanion concret se fac cu ajutorul indicelui elementului prin care el este implementat, asociindu-se identificatori sugestivi fiecărui indice. Este practic ca valoarea unui asemenea identificator să fie stabilit printr-o funcţie care asigură alocarea elementelor tabloului de fanioane. O vom denumi *f_creat ()*. Disocierea unui identificator de elementul cu care este pus în legătură prin funcţia *f_creat ()* se face cu ajutorul unei funcţii de dezalocare, denumit *f_destroy ()*.

În fine, pentru iniţializarea tabloului *_flg* [], mecanismul de excludere mutual prin fanioane de excluziune active dispune de o funcţie denumită *f_init ()*.

În figura 6, este redat, sintetic, ansamblul celor cinci funcţii prin care utilizatorul poate face referiri la fanioane.

```
        void f_init (void)
/*      iniializeaz tabloul de fanioane                    */

        usshort f_creat (void)
/*      creează un fanion, furnizînd indicele              */
/*      elementului                                        */
/*      tabloului _flg [] pe care îl aloc ă în acest sens  */

        void f_destroy (usshort ind_flg)
/*      distruge un fanion, dezalocînd elementul aferent   */
/*      din tabloul _flg []; argumentul ind_flg reprezintă */
/*      indicele respectivului element                     */
unsigned f_wait (usshort ind_flg, usshort timeout)
/*      asigură, dacă este cazul printr-o ateptare         */
/*      limitat superior de argumentul timeout, ca         */
/*      procesul curent să-şi poată iniţia secţiunea critică */
/*      referitoare la resursa asociată fanionului cu      */
/*      indicele ind_flg;dacă aşteptarea depăşeşte         */
/*      limita, funcţia returnează o valoare unsigned     */
/*      nenulă; altfel, valoarea returnată este nulă       */

        void f_signal (usshort ind_flg)
/*      semnalează încheierea unei secţiuni critice        */
/*      referitoare la resursa asociată fanionului cu      */
/*      indicele ind_flg, deblocînd toate procesele aflate */
/*      în coada acestuia, şi predă controlul dispecerului */
```

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

Figura 6. Funcţiile aferente mecanismului de excludere mutuală prin *fanioane de excluziune active.*

Funcţia *f_init ()*

Această funcţie are rolul de a iniţializa tabloul de fanioane *_flg* [], prin stabilirea valorii *NULL* pentru componentele *pie* ale tuturor elementelor sale.

Textul C al funcţiei *f_init ()* este următorul:

```
1   void f_init (void)
2   {
3   register FLAG *fg;
4   register unsigned j;
5   fg=&_flg[0];
6   for (j=0;j<MAX_FLG;j++) {
7   fg->pie=NULL;
8   fg++;
9   }
10 }
```

Figura 7. Textul funcţiei *f_init ()*.

Funcţia *f_creat ()*

Funcţia *f_creat ()* are rolul de a identifica un element liber al tabloului de fanioane *_flg* [] şi de a-l aloca, returnând indicele-i.

De asemenea, funcţiei *f_creat ()* îi revine sarcina de a poziţiona pointerul *pie* al fanionului pe care îl alocă pe primul element al cozii şi de a iniţializa la valoarea "0" componenta *f* a acestui fanion.

Textul funcţiei *f_creat ()* este redat în figura 8.

```
1   usshort f_creat (void)
2   {
3   register FLAG *fg;
4   register unsigned j;
5   fg=&_flg[0];
6   j=0;
7   _lock_ ();
8   while ((fg->pie)&&(j<MAX_FLG)) {
9     fg++;
10    j++;
```

Figura 8. Textul funcţiei *f_creat ()*.
-partea întîi-

```
11  /* în ciclul while, s-a considerat ca semn     */
12  /* distinctiv pentru un fanion încă nealocat */
13   /* valoarea NULL a componentei sale pie      */
14  /* (vezi funcţile f_init () i f_destroy ())    */
15   if (jAX_FLG) {
16        fg->pie=&(fg-.coada[0]);
17        /* se pozitionează pie pe      */
18        /* primul element al cozii     */
19        fg->f=0;
20   }
21   else {
22        /* se generează un mesaj de eroare    */
23   }
24  _unlock_();
25  return (j);
26 }
```

Figura 8. Textul funcţiei *f_creat ()*.
-partea a doua-

Funcţia *f_destroy ()*

Funcţia *f_destroy ()* are rolul de a dezaloca elementul tabloului de fanioane *_flg* [] cu indicele specificat prin argumentul ei, atribuind valoarea *NULL* componentei *pie* a acestui element.

Acţiunea de distrugere a funcţiei *f_destroy ()* se exercită doar dacă ea găseşte coada fanionului vid. Altfel, funcţia se rezumă la generarea unui mesaj de eroare.

Textul funcţiei *f_destroy ()* este redat în figura 9.

```
1   void f_destroy (usshort ind_flg)
2  {
3   register FLAG *fg;
4   fg=&_flg[ind_flg];
5    _lock_();
6   if (fg->pie!=&(fg->coada[0])) {
7   /* eroare: coada fanionului este nevidă   */
8   }
9   else {
10  fg-pie=NULL;
11  }
12   _unlock_();
13  }
```

Figura 9. Textul funcţiei *f_destroy ()*.

Funcţia *f_wait ()*

Funcţia *f_wait ()* are rolul de a efectua operaţia *TAS* asupra componentei *f* a fanionului cu indicele *ind_flg* şi de a permite procesului care o execută iniţierea secţiunii sale critice, dacă această operaţie găseşte componenta *f* la valoarea "0", respectiv de a bloca acest proces şi de a-l înscrie în coada fanionului, în caz contrar.

Operaţia de blocare a procesului în cauză se execută prin apelul funcţiei *sleep ()*. Acesteia i se pasează, în momentul apelului, valoarea argumentului *timeout*, prin care se precizează timpul limită cât procesul poate rămâne blocat. Dacă la expirarea acestui timp, procesul în cauză este încă blocat, se va produce deblocarea lui automat. O asemenea deblocare denotă o anomalie, care trebuie tratat în mod corespunzător de către utilizator. În sprijinul acestuia, *funcţia f_wait ()* returnează o valoare *unsigned*, nenulă, în cazul în care anomalia s-a petrecut, respectiv nulă, altfel.

Textul funţciei *f_wait()* este următorul:

```
1   unsigned f_wait (usshort ind_flg, usshort timeout)
2  {
3   register FLAG *fg;
4   unsigned aux;
5   fg=&_flg[ind_flg];
6   _lock_();
7   while ((fg->f)&&(!aux)) {
8         *fg->pie=_task_crt;
9         fg-.pie++;
10        aux=sleep (timeout);
11  }
12  if (!aux) {
13        fg-f=1;
14  }
15  _unlock_();
16  return (aux);
17 }
```

Figura 10. Textul funcţiei *f_wait ()*.

Funcţia *f_signal ()*

Funcţia *f_signal ()* are rolul de a efectua operaţia *RES* asupra componentei *f* a fanionului cu indicele *ind_flg* i de a debloca toate procesele aflate în coada acestui fanion. Deblocarea proceselor se asigură prin apelul funcţiei *wakeup ()*. Dacă nici un proces nu este găsit în coadă, *funcia f_signal ()* asigură rămânerea în rulare a procesului care o execută, nemaifăcând apel la dispecer.

Textul funcţtiei *f_signal ()* este redat în figura 11.

```
1   void f_signal (usshort ind_flg)
2  {
3   register FLAG *fg;
4   register usshort tsk;
5   fg=&_flg[ind_flg];
6   _lock_();
7   while (fg->pie!=&(fg-coada[0])) {
8   fg->pie--;
9    tsk=*(fg->pie);
10   wakeup (tsk);
11  }
12   _unlock_();
13 }
```

Figura 11. Textul funcţiei *f_signal ()*.

4. Concluzii

Mecanismul de excludere mutuală propus este avantajos mai ales în cazul secţiunilor critice cu durata mai mică decât perioada de comutare a proceselor şi referitoare la resurse critice puternic disputate [3].

Într-un asemenea caz, presupunând o dispecerizare prin rotaţie, dacă un proces îi iniţiază o secţiune critică referitoare la o resursă cu puţin înaintea momentului unei comutări, atunci, cu ocazia comutării, el va pierde controlul asupra procesorului, împiedicând, însă, alte procese să acceseze resursa în cauză. Drept urmare, acestea se vor bloca, rând pe rând. La un moment, reprimind controlul, primul proces îşi va încheia, relativ rapid, secţiunea critică antamată mai devreme.

În ipotezele considerate, ar fi avantajos ca toate procesele blocate la resursa respectivă să fie deblocate, deodată, pentru că, în mod normal, cuantumul de timp ce va fi alocat fiecăruia dintre ele pentru o repriză de rulare va fi suficientă pentru parcurgerea completă a secţiunii critice a fiecăruia, într-o singură astfel de repriză. Ori, niciunul dintre mecanismele consacrate de excludere mutuală nu asigură acest lucru. Mecanismul propus, însă, o face.

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 Reèl, Dunod, Paris, 1990.