SISTEMUL DE OPERARE DE TIMP REAL μCOS
(Micro Controller Operating System)

Scopul lucrării :

- Descrierea funcţiilor unui sistem de operare de timp real
- Rezolvarea problemelor specifice operării cu mai multe procese (multitasking) : accesul exclusiv la resurse comune, sincronizarea şi comunicaţia între procese
- Exemplificarea utilizării primitivelor sistemului de timp real μCOS

1. Introducere. Notiuni elementare despre sistemele de operare de timp real

Un sistem de operare de timp real reprezintă ansamblul de programe de sistem (primitive) ce asigură următoarele funcţii într-un sistem de timp real :

- execuţia corectă a tuturor proceselor (sarcinilor) sistemului de timp real
- asigurarea unui timp de răspuns strict corelat cu cerinţe de timp impuse din afara sistemului de timp real
- posibilitatea de execuţie "simultană" (cu paralelism virtual) a unor procese secvenţiale (**task-uri**) care cooperează şi rulează pe principiul distribuirii timpului de execuţie al UCP între procese
- realizarea protecţiei resurselor comune ale sistemului de timp real
- realizarea comunicaţiei între task-uri
- sincronizarea proceselor secvenţiale
- interfaţarea dintre sistemul de timp real şi utilizator (cu ajutorul unor periferice standard şi/sau cu ajutorul unor periferice nestandard ca de exemplu convertoare analog-numerice şi numeric- analogice.

Aceste primitive constituie nucleul (**kernel**) sistemului de operare de timp real.
Un sistem de operare de timp real este un sistem **multitasking** : un utilizator poate executa "simultan" mai multe procese.
Totodată un sistem de operare de timp real poate fi şi un sistem **multiutilizator** : mai mulţi utilizatori pot lucra independent pe sistemul de timp real ca şi cum ar avea întreg sistemul la dispoziţie.

2. Funcţiile sistemului de operare de timp real μCOS

Funcţiile (primitivele) sistemului de operare μCOS sînt următoarele :

- crearea de procese (task-uri)
- modificarea priorităţii procesului curent în execuţie
- distrugerea task-ului curent în execuţie
- planificarea proceselor (task-urilor)
- asigurarea comunicării şi sincronizării între procese (prin cutii poştale şi cozi de mesaje)
- asigurarea accesului exclusiv la resurse comune (prin semafoare generalizate)

Primitiva de creare a unui task realizează înregistrarea la sistemul de operare a noului task prin adăugarea acestuia într-o listă de task-uri ce conține cîte un bloc de control al task-ului (*Task Block Control* - TCB) pentru fiecare task în parte. Blocul de control este inițializat la crearea task-ului.

Primitiva de creare a unui task este :

OS TaskCreate (Nume_task, (void *) &Date, (void *) &stack(size_s),int p)

unde *Nume_task* este numele task-ului, *Date* este zona de date a task-ului , *stack* este stiva asociată task-ului (de dimensiune *size_s*) , iar *p* este prioritatea asociată task-ului.

Distrugerea task-ului curent presupune eliminarea din aceasta lista a procesului in execuție la initiativa acestuia.

Primitiva de distrugere a unui task este :

OSTaskDelete (void)

- distruge procesul curent în execuție; modifică lista de procese (scoate procesul distrus şi structura de date a planificatorului)

Modificarea priorității se efectuează tot la initiativa proprie a task-ului curent în executie.

Primitiva de modificare a priorității este :

OSChangePrio (int newp)

- *newp* este noua prioritate (între 0 şi 63).

Task-urile se autoplanifică; sistemul de operare măsoară timpul de execuție pentru fiecare task si actualizează un contor din blocurile de control ale proceselor.

3. Structura de date a sistemului μCOS

3.1 Blocul de control al task-ului (TCB)

Are structura ilustrată în figura 1.

Informațiile conținute de TCB, pentru fiecare task în parte sînt :

- un pointer către stiva asociată procesului
- starea procesului (gata de execuție - *ready* , suspendat, suspendat la semafor, suspendat în aşteptarea unui mesaj din cutia poştală sau suspendat în aşteptarea unui mesaj din coada de mesaje)
- prioritatea task-ului
- informație de legătură cu TCB-urile asociate task-urilor vecine

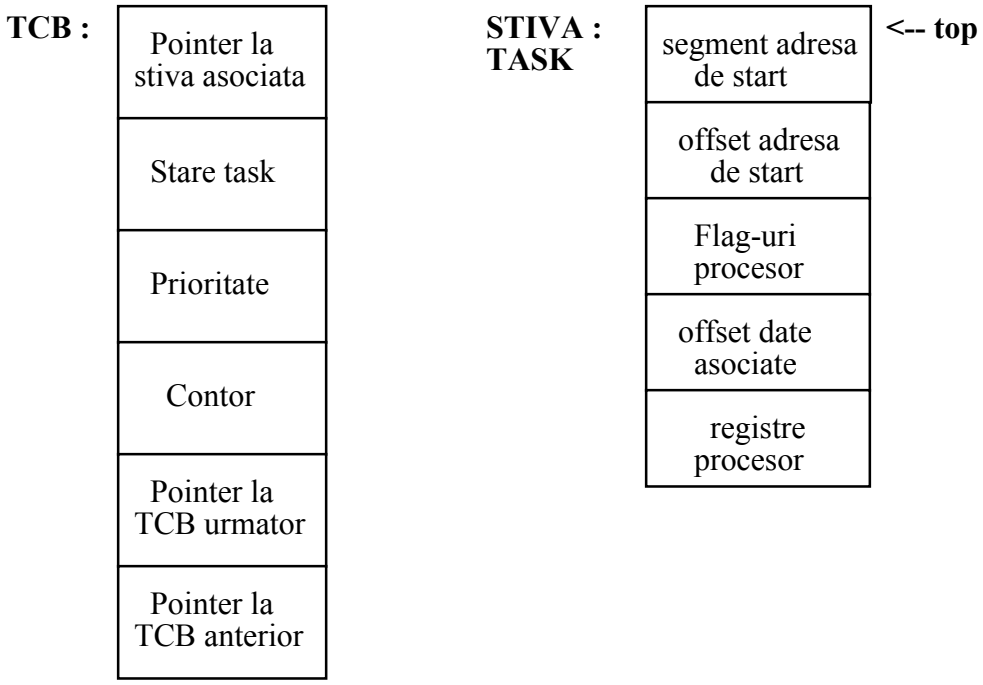


Figura 1. Structura TCB; stiva asociată procesului

Conținutul TCB poate fi modificat numai la crearea unui task sau la distrugerea acestuia.

3.2. Structura de date asociată planificatorului de procese

Procesele sînt împărțite în 8 grupuri (numerotate de la 0 la 7); fiecare grup cuprinde pînă la 8 procese ierarhizate după 8 nivele de prioritate.

Structura de date a planificatorului cuprinde un octet *OSRdyGrp* şi un tabel de 8 octeți *OSRdyTbl* ca în figura 2 .

Prioritatea unui task descreşte de la 0 (task-ul de prioritate maximă) pînă la 63 (task-ul de prioritate minimă).

Se pot defini pînă la 64 de task-uri, fiecare de prioritate distinctă.

În figura 2 există task-urile de prioritate : 0, 18,21,56 şi 63. Grupurile active sînt: 0, 2 şi 7.

Octetul *OSRdyGrp* indică grupul de procese active. Introducerea acestui octet permite o căutare a proceselor gata de execuție mai rapidă (nu se mai parcurge toată tabela *OSRdyTbl*, ci doar liniile corespunzătoare grupului activ).

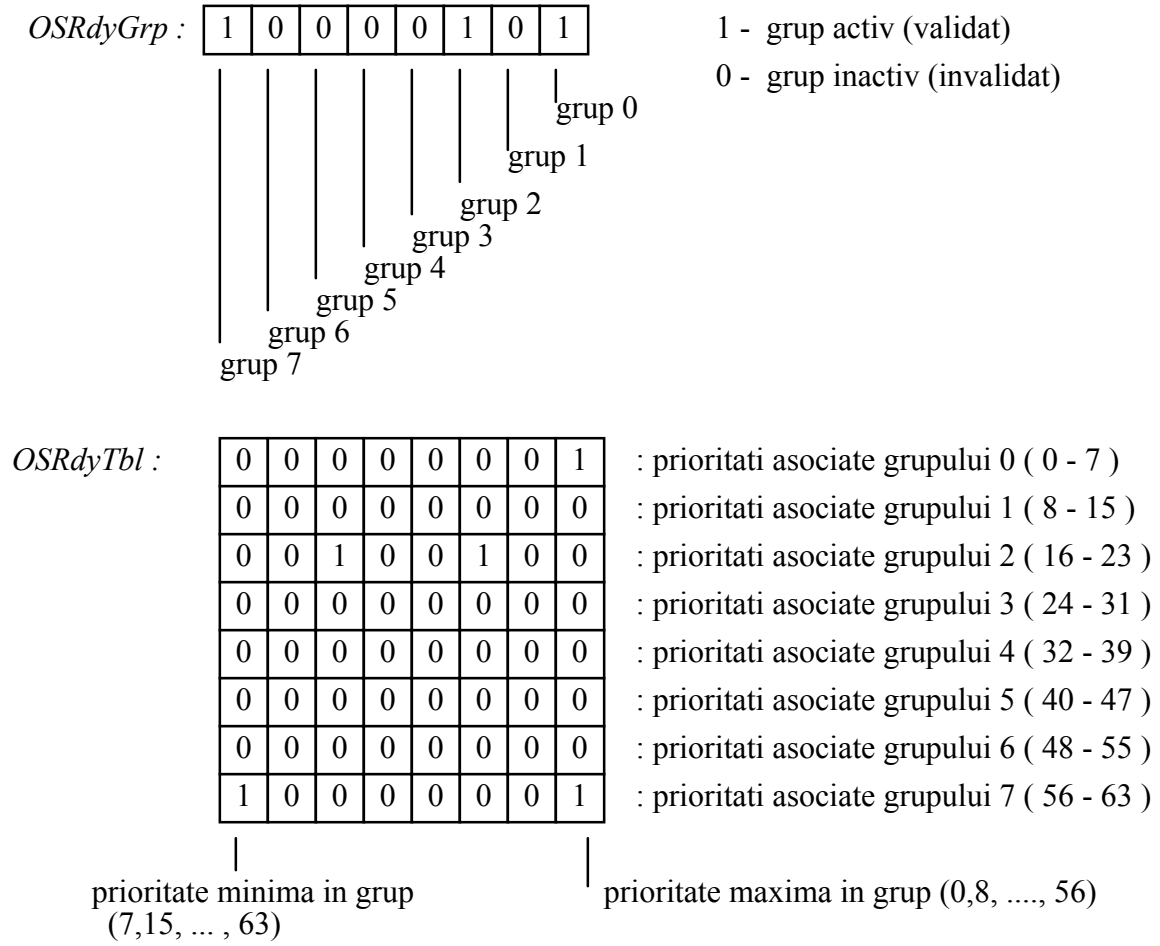


Figura 2. Structura de date a planificatorului din cadrul sistemului de operare □COS

Algoritmul de planificare caută in structura de date procesul de prioritate maximă şi îl pune în execuție.

3.3. Structura de date asociată semafoarelor

Sistemul de operare □COS utilizează semafoare generalizate pentru a proteja resursele comune. Structura de date asociată unui semafor generalizat cuprinde un contor (*OSSemCnt*) şi informație de identificare a procesului (*OSSemGrp* şi *OSSemRdy*) ce aşteaptă autorizația de trecere pe lîngă semafor (similară informației utilizate de planificator).

Structura de date pentru un semafor generalizat este ilustrată în figura 3.

Contorul *OSSemCnt* este un cuvînt pe 16 biți pozitiv sau negativ.

Contorul poate fi modificat de către primitive specifice sistemului □COS asfel : atunci cînd un proces doreşte acces la resursa protejată de semafor contorul este decrementat, iar atunci cînd un proces eliberează resursa contorul este incrementat.

Valoarea zero a acestui contor indică faptul că resursa protejată de semafor poate fi ocupată; o valoare negativă a acestui contor indică numărul proceselor ce aşteaptă la semafor.

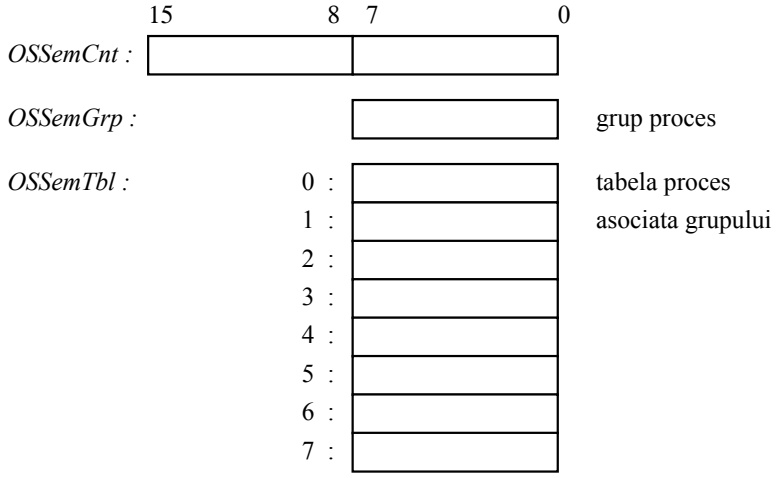


Figura 3. Structura unui semafor generalizat

Informația de identificare a procesului care utilizează semaforul este compusă dint-un octet *OSSemGrp* şi un tablou de 8 octeți *OSSemTbl*, similare cu *OSRdyGrp* şi respectiv *OSRdyTbl* din structura de date a planificatorului.

3.4. Structura de date asociată cutiilor poştale

Cutiile poştale sînt utilizate, de către sistemul de operare □COS, pentru a realiza comunicația şi sincronizarea între procese . Structura de date asociată unei cutii poştale cuprinde un pointer către mesajul depus (care trebuie citit) *OSMboxMsg* şi informație de identificare a procesului care utilizează cutia poştală dată de *OSMboxGrp* şi *OSMboxTbl* (similară celei folosite de planificator şi semafoare).

Această structură de date este ilustrată în figura 4.

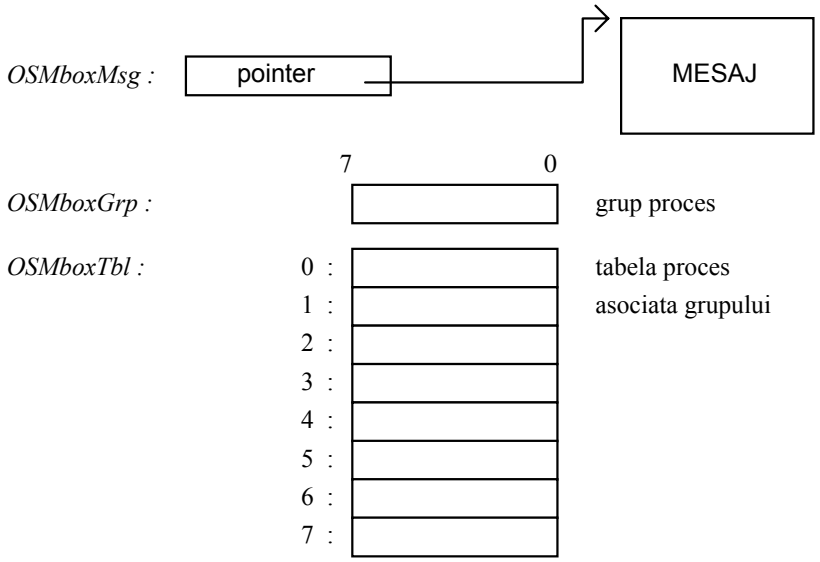


Figura 4. Structura de date a cutiei poştale

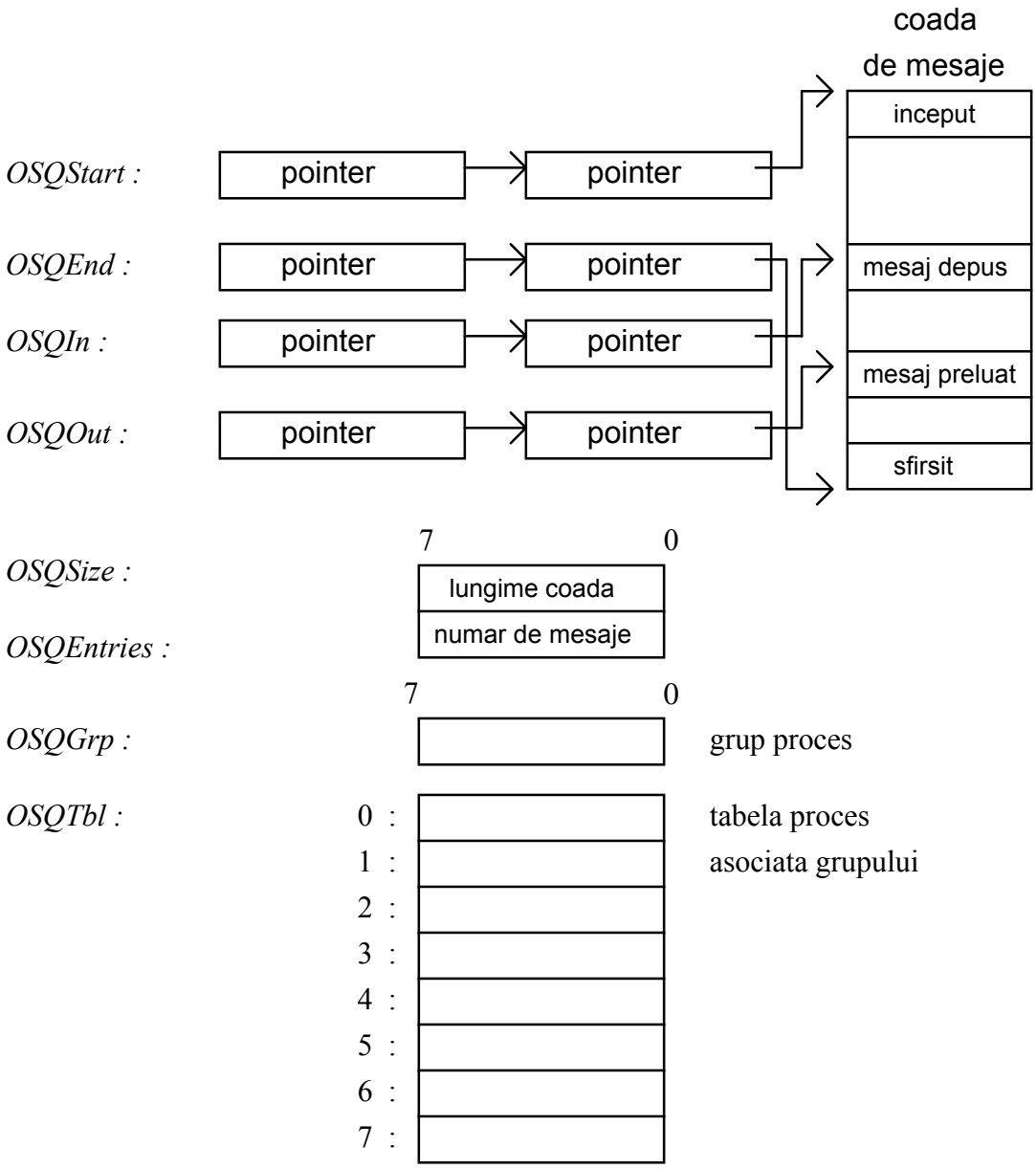
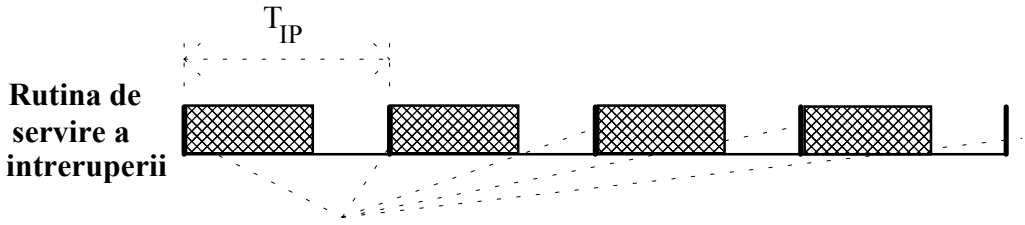


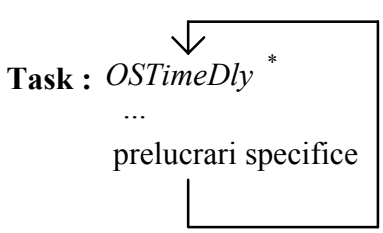
Figura 5. Structura de date asociată cozilor de mesaje

4. Planificarea proceselor în sistemul de operare µCOS

Procesele definite (create) sînt executate în ordinea priorității asociate dacă contorul din TCB-ul propriu este zero.Prin intermediul acestui contor task-urilr se autoplanifică pentru execuție; variabila contor este decrementată , într-o rutină de întreruperi periodice, pentru toate procesele create.

Procesele sînt puse în execuție după următoarea schemă (figura 6).





* - sau una din primitivele
OSSemPend, OSMBoxPend, OSQPend

*OSTimeDly** : actualizeaza TCB.contor (autoplanificare)
suspenda temporar procesul (modifica structura de date a planificatorului)
apel *OSSched*
.....
RET

OSSched : alege procesul de prioritate maxima cu TCB.contor = 0
(cautare in structura de date asociata planificatorului)
salveaza contextul procesului in stiva asociata
pune in executie noul proces
RET

OSTimeTick : Pentru toate procesele create **executa**

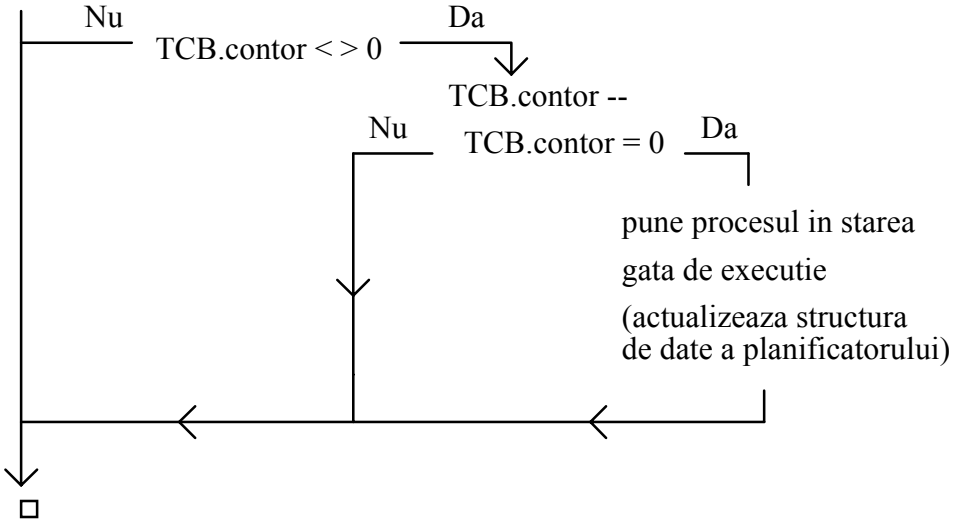


Figura 6. Modul de planificare pentru sistemul de operare de timp real μCOS

Task-urile trebuie să reprezinte bucle infinite care să conțină aplelul primitivei *OSTimeDly*; această primitivă permite autoplanificarea procesului. Dacă un proces nu apelează această primitivă sau valoarea TCB.contor=0, atunci se va executa continuu (dacă ajunge cel puțin o dată în execuție). Primitiva *OSTimeDly* poate fi înlocuită cu una din primitivele *OSSemPend*, *OSMboxPend* sau *OSQPend* care realizează autoplanificarea în mod similar.

Sînt necesare întreruperi periodice care să pună în execuție (în rutina de servire) primitiva *OSTimeTick* ; această primitivă asigură actualizarea structuri de date a planificatorului (evidența proceselor gata de execuție) conform variabilei TCB.contor.

Primitiva *OSTimeDly* trebuie apelată astfel :

OSTimeDly(int val_contor);

unde val_contor este valoarea ce se va atribui lui TCB.contor

5. Utilizarea semafoarelor în sistemul µCOS

Există 3 primitive asociate lucrului cu semafoare :

OSSemInit(OS_SEM *sem, int cnt) ;

-inițializează structura de date de tip semafor (OS_SEM ca în secțiunea 3.3), valoarea contorului este *cnt*, iar restul cîmpurilor sînt zero.

OSSemPend(OS_SEM *sem, int time_out) ;

- decrementează contorul semaforului
- cere autorizația de trecere pe lîngă semafor
- dacă autorizația nu s-a obtinut (contor semafor < > 0) atunci pune procesul apelant în starea de asteptare la semafor (prin modificarea structurii de date a planificatorului)
- planifică procesul pentru execuție peste time_out cuante de timp (cînd se va realiza o nouă încercare de trecere pe lîngă semafor).
- pune în execuție următorul proces (apel *OSSched*)

OSSemPost(OS_SEM *sem) ;

- incrementează contorul semaforului (eliberează resursa)
- suspendă procesul curent (modifică structura de date a planificatorului)
- pune în execuție următorul proces (apel *OSSched*)

6. Utilizarea cutiilor poştale în sistemul µCOS

Există 3 primitive asociate lucrului cu cutiile poştale :

OSMBoxInit(OS_MBOX *mbox, void *msg) ;

-inițializează structura de date de tip cutie poştală (OS_MBOX ca în secțiunea 3.4), pointer-ul la mesaj este *msg*, iar restul cîmpurilor sînt zero.

void **OSMBoxPend*(OS_MBOX *mbox, int timeout, int *err) ;

- citeşte mesajul din cutia poştală (dacă acesta există)
- dacă mesajul nu există (*msg* este pointer nul) atunci pune procesul apelant în starea de asteptare mesaj în cutia poştală (prin modificarea structurii de date a planificatorului)
- planifică procesul pentru execuție peste *timeout* cuante de timp (cînd se va realiza o nouă citire a cutiei poştale).
- pune în execuție următorul proces (apel *OSSched*)
- întoarce , la adresa &err , un cod de eroare (dacă mesajul nu există)
- întoarce un pointer către mesajul citit

OSSemPost(OS_MBOX *mbox) ;

- scrie mesaj în cutia poştală
- suspendă procesul curent (modifică structura de date a planificatorului)
- pune în execuție următorul proces (apel *OSSched*)

7. Utilizarea cozilor de mesaje în sistemul μCOS

Există 3 primitive asociate lucrului cu cozile de mesaje :

OSQInit(OS_Q *q, void **start, int size) ;

-inițializează structura de date de tip coadă de mesaje (OS_Q ca în secțiunea 3.5), pointer-ul la inceputul cozii este *start*, pointer-ul la sfîrşitul cozii este determinat de *start* şi lungimea cozii *size* (care se actualizează corespunzător), iar restul cîmpurilor sînt zero.

void **OSQPend*(OS_Q *q, int timeout, int *err) ;

- citeşte primul mesaj din coada de mesaje (dacă există)
- dacă nu există mesaj (coadă de mesaje vidă) atunci pune procesul apelant în starea de asteptare mesaj în coada de mesaje (prin modificarea structurii de date a planificatorului)
- planifică procesul pentru execuție peste *timeout* cuante de timp (cînd se va realiza o nouă citire a cozii de mesaje).
- pune în execuție următorul proces (apel *OSSched*)
- întoarce , la adresa &err , un cod de eroare (dacă coada e vidă)
- întoarce un pointer către mesajul citit

OSQPost(OS_Q *q, void *msg) ;

- scrie mesaj în coada de mesaje
- suspendă procesul curent (modifică structura de date a planificatorului)

- pune în execuție următorul proces (apel *OSSched*)

8. Exemple

A. Planificarea execuţiei proceselor

Programul de la exemplul A crează 4 procese care se autoplanifică pentru execuție cu diferite cuante de timp. Procesul *Task3* se poate distruge după ce s-a executat de un anumit număr de ori.

Cuanta de timp este de 55ms (se generează intreruperi periodice de la ceasul de tim real al PC-ului).

```
/*
*********************************************************************
*                    TEST11.C
*                  uCOS EXAMPLE
*********************************************************************
*/

#include   <STDIO.H>
#include   <STRING.H>
#include   <CTYPE.H>
#include   <STDLIB.H>
#include   <CONIO.H>
#include   <DOS.H>
#include   "UCOS186C.H"
#include   "UCOS.H"
extern void interrupt (*OldTickISR)(void);//rutina de intreruperi 18ms


#define        OS_MAX_TASKS    10 //numar maxim de tak-uri
#define        STK_SIZE      2048 //dimensiunea stivei

#define                    nr_c            3         //nr. cuante pt. autosuspendare


OS_TCB         OSTCBTbl[OS_MAX_TASKS]; //tabela de control a task-urilor
UWORD           OSIdleTaskStk[STK_SIZE];//task fals
UWORD           Stk1[STK_SIZE]; //definire stive task-uri
UWORD           Stk2[STK_SIZE];
UWORD           Stk3[STK_SIZE];
UWORD           Stk4[STK_SIZE];
UWORD           Stk5[STK_SIZE];

char           Data1[5] = "C";//definire date proprii task-urilor
char           Data2[5] = "P1|";
char           Data3[5] = "P2|";
char           Data4[5] = "P3|";
```

```
char          Data5[5] = "P4|";

void far      Task_Afis(void *data); //definire task-uri
void far      Task1(void *data);
void far      Task2(void *data);
void far      Task3(void *data);
void far      Task4(void *data);

void interrupt (*OldTickISR)(void);
int i=1;          //contor de cuante
int j=0;          //contor de autosuspendare task 3

void main(void)
{
   UBYTE err;


   clrscr();
   //salveaza vechiul vector de intrerupere (nivelul 0x08)
   //intreruperi ceas de timp real 18ms
   OldTickISR = getvect(0x08);
   //seteaza intreruperea software UCOS=0xF1 pentru comutarea contextului
   setvect(UCOS, (void interrupt (*)(void))OSCtxSw);
   setvect(0xF2, OldTickISR);
   //seteaza intreruperea software 0xF2 pentru executia rutinei
   //vechi de pe nivelul 0x08
   //initializeaza (creaza) un task fals (idle)
   OSInit(&OSIdleTaskStk[STK_SIZE], OS_MAX_TASKS);
   //creaza task-urile
   OSTaskCreate(Task_Afis, (void *)&Data1, (void *)&Stk1[STK_SIZE], 1);
   OSTaskCreate(Task1, (void *)&Data2, (void *)&Stk2[STK_SIZE], 2);
   OSTaskCreate(Task2, (void *)&Data3, (void *)&Stk3[STK_SIZE], 3);
   OSTaskCreate(Task3, (void *)&Data4, (void *)&Stk4[STK_SIZE], 4);
   OSTaskCreate(Task4, (void *)&Data5, (void *)&Stk5[STK_SIZE], 5);

   //start executie
   OSStart();
}

void far Task_Afis(void *data)
{
   setvect(0x08, (void interrupt (*)(void))OSTickISR);
   while (1) {
         OSTimeDly(18);//nu se va modifica : afisare la o secunda
         printf("\n");
         printf("%s",data);
         printf("%d:|",i);
```

```
        i++;    //nr. cuantei
        if(kbhit()){
        switch(getch()){
        case 'x':
        case 'X':
           {
           setvect(0x08, OldTickISR);
           exit(0);
           break;
           }
        }
        }
   }
}
//planificarea proceselor in cuante = multiplii de 18 (in secunde)

void far Task1(void *data)
{
   while (1) {

        OSTimeDly(18);              //1 sec
        printf("%s",data);
            }
}

void far Task2(void *data)
{
   while (1) {
        OSTimeDly(36);              //2 sec
        printf("%s",data);
             }
}

void far Task3(void *data)
{
   while (1) {
        OSTimeDly(54);              //3 sec
        printf("%s",data);
        j++;
        if(j==nr_c) OSTaskDelete();
            }
}

void far Task4(void *data)
{
   while (1) {
        OSTimeDly(72);              //4 sec
        printf("%s",data);
```

```
        }
}
```

B. Comunicarea între procese şi utilizarea semafoarelor

Programul de la exemplul B crează următoarele procese :

- un proces de afişare *DispTask* (afişează variabilele protejate Ctr1,Ctr2,Min şi Sec)
- un proces de citire a tastaturii *KeyTask* (citeşte tastatura şi în funcție de tasta apăsată înscrie mesaje în cutia poştală sau în coada de mesaje)
- un proces *Task1* (verifică dacă există mesaj în cutia poştală şi incrementează variabila protejată *Ctr1*). Afisează mesajul din cutia postală.
- un proces *Task2* (verifică dacă există mesaje în coada de mesaje şi incrementează variabila protejată *Ctr2*). Afişează mesajele din coada de mesaje.
- un proces *Task3* (incrementează variabilele protejate *Sec* şi *Min*)

Există 4 variabile considerate resurse comune : *Sec*, *Min* ,*Ctr1* şi *Ctr2* protejate prin semafoare.

Cuanta de timp este de 55ms (se generează intreruperi periodice de la ceasul de tim real al PC-ului).

```
/*
*********************************************************
*                    TEST21.C
*                  uCOS EXAMPLE
*********************************************************
*/

#include   <STDIO.H>
#include   <STRING.H>
#include   <CTYPE.H>
#include   <STDLIB.H>
#include   <CONIO.H>
#include   <DOS.H>
#include   "UCOS186C.H"
#include   "UCOS.H"
extern void interrupt (*OldTickISR)(void);


#define        OS_MAX_TASKS    10    //numar maxim de task-uri
#define        STK_SIZE      1024    //dimensiunea stivei
#define        Q_SIZE          10    //dimensiunea cozii de nesaje

OS_TCB         OSTCBTbl[OS_MAX_TASKS];//tabela de task-uri
```

```
UWORD          OSIdleTaskStk[STK_SIZE];//task fals

UWORD          KeyStk[STK_SIZE]; //stive asociate task-urilor
UWORD          DispStk[STK_SIZE];
UWORD          Task1Stk[STK_SIZE];
UWORD          Task2Stk[STK_SIZE];
UWORD          Task3Stk[STK_SIZE];

OS_SEM         Sem; //structura de semafor
OS_MBOX          Mbox;  //structura de cutie postala
OS_Q           Q;     //structura de coada de mesaje

void          *QData[Q_SIZE]; //coada de mesaje

UWORD          Ctr1 = 0;       //structura de date protejata
UWORD          Ctr2 = 0;       //prin semafoare
UBYTE          Min  = 0;
UBYTE          Sec  = 0;

void far       KeyTask(void *data);    //definirea task-urilor
void far       DispTask(void *data);
void far       Task1(void *data);
void far       Task2(void *data);
void far       Task3(void *data);

void interrupt (*OldTickISR)(void);     //rutina de intrerupere 18ms

char mesaj[80];        //variabile auxiliare
char mesaj1[20][80];

int cnt=0;
int cnt1=0;

void main(void)

{
   clrscr();
   //seteaza intreruperile (ca in exemplul A)
   OldTickISR = getvect(0x08);
   setvect(UCOS, (void interrupt (*)(void))OSCtxSw);
   setvect(0xF2, OldTickISR);
   //defineste (creaza) un task fals (idle)
   OSInit(&OSIdleTaskStk[STK_SIZE], OS_MAX_TASKS);
   //initializeaza semaforul
   OSSemInit(&Sem, 1);
   //initializeaza cutia postala
   OSMboxInit(&Mbox, (void *)0);
   //initializeaza coada de mesaje
```

```
   OSQInit(&Q, &QData[0], Q_SIZE);
   //creaza task-urile
   OSTaskCreate(DispTask, (void *)0, (void *)&DispStk[STK_SIZE],   0);
   OSTaskCreate(KeyTask,  (void *)0, (void *)&KeyStk[STK_SIZE],    1);
   OSTaskCreate(Task1,    (void *)0, (void *)&Task1Stk[STK_SIZE], 10);
   OSTaskCreate(Task2,    (void *)0, (void *)&Task2Stk[STK_SIZE], 20);
   OSTaskCreate(Task3,    (void *)0, (void *)&Task3Stk[STK_SIZE], 30);
   //start executie
   OSStart();
}

void far KeyTask(void *data)
{
   UBYTE i;

   setvect(0x08, (void interrupt (*)(void))OSTickISR);
   while (1) {
         OSTimeDly(1);
         if (kbhit()) {
            switch (getch()) {
                  case '1': cnt++;
                              sprintf(mesaj,"Mesaj in cutia postala [%d] ",cnt);
                              OSMboxPost(&Mbox, (void *)mesaj);
                              break;
                  case '2': cnt1++;
                              sprintf(mesaj1[cnt1],"Mesaj in coada [%d]",cnt1);
                              OSQPost(&Q, (void *)mesaj1[cnt1]);
                              break;
                  case 'x':
                  case 'X': setvect(0x08, OldTickISR);
                              exit(0);
                              break;
            }
         }
   }
}


void far Task1(void *data)
{
   UBYTE err;
   void* m;

   while (1) {
         m=OSMboxPend(&Mbox, 36, &err);
         gotoxy(1,1);
         if (m!=NULL) printf("%s\n",(char*)m);
```

```
        OSSemPend(&Sem, 0);
        Ctr1++;
        gotoxy(1, 13);
        printf("Ctr1-task1  =  %4d", Ctr1);
        OSSemPost(&Sem);
    }
}


void far Task2(void *data)
{
    UBYTE err;
    void *m;
    int k;
    k=1;

    while (1) {
        m=OSQPend(&Q, 72, &err);
        gotoxy(50,k);
        if (m!=NULL) {printf("%s\n",(char*)m);k++;if(k==11) k=1;}
        OSSemPend(&Sem, 0);
        Ctr2++;
        gotoxy(1, 14);
        printf("Ctr2-task1  =  %4d", Ctr2);
        OSSemPost(&Sem);
    }
}


void far Task3(void *data)
{
    while (1) {
        OSTimeDly(18);
        OSSemPend(&Sem, 0);
        Sec++;
        if (Sec > 59) {
           Sec = 0;
           Min++;
        }
        gotoxy(1, 15);
        printf("Min:Sec -task3  =  %02d:%02d", Min,Sec);
        OSSemPost(&Sem);
    }
}


void far DispTask(void *data)
{
```

```
   UWORD ctr1, ctr2;
   UBYTE min,  sec;




while (1) {
        OSTimeDly(6);

        OSSemPend(&Sem, 0);
        ctr1 = Ctr1;
        ctr2 = Ctr2;
        min  = Min;
        sec  = Sec;
        OSSemPost(&Sem);
        gotoxy(1, 9);
        printf("Clock = %02d:%02d", min, sec);
        gotoxy(1, 10);
        printf("Ctr1  =  %4d", ctr1);
        gotoxy(1, 11);
        printf("Ctr2  =  %4d", ctr2);
   }
}
```

9. Desfăşurarea lucrării

a) Se vor studia primitivele nucleului de timp □COS, structurile de date asociate planificatorului, cutiilor poştale, cozilor de mesaje şi semafoarelor.

b) Pentru exemplul A se va studia modul de creare, planificare şi distrugere a proceselor secvențiale .

c) Pentru exemplul B se va studia modul de utilizare a cutiei poştale, a cozii de mesaje şî a semaforului.

Exemplificarea programelor din secțiunea 8 presupune deschiderea unui **proiect** în C astfel :

- din mediul integrat Borland C se selectează comanda **Project** , subcomanda *Open Project*

- se alege un nume de proiect (**Test11.prj** pentru exemplul A şi **Test21.prj** pentru exemplul B) sau se tastează un nume nou (pentru un proiect nou)). Programele de aplicație sînt TEST11.C - pentru **Test11.prj** şi TEST21.C - pentru **Test21.prj**.

- pentru un proiect nou se adaugă (cu comanda *Add*) fişierele : UCOS.C, UCOS186.C şi UCOS186A.ASM împreună cu programul de aplicație TEST11.C sau TEST21.C care se vor modifica). După adăugarea tuturor fişierelor se va da comanda *Done*.

- Compilarea proiectului se face cu comanda **F9**, iar execuția cu **Ctrl F9**.