Beágyazott információs rendszerek

Szepessy Zsolt (zszepes@mit.bme.hu)

Beágyazott rendszerek alapkomponensei II. : Software

Beágyazott rendszerek SW architektúrái

A korábbi ütemezési algoritmusok megvalósítását a processzor rendszerint szekvenciális műveletvégrehajtási tulajdonságához kell igazítani. A processzoridőt hatékonyan ki kell használni. A valós-idejű előírások a rendszerrel szemben támasztott legfontosabb követelmények közé tartoznak.

Elemzés szempontjai:
- Események legrosszabb esetben számítható kiszolgálási ideje.
- Kiszámítható működés.
- Hatékonyság.
- Fejleszthetőség.
- Alkalmazási kör.

A beágyazott rendszerek alapvető *gyakorlati* ütemezési módszereit az alábbi csoportokba sorolhatjuk:
1. Periódikus ütemezés: körforgó ütemezés, súlyozott körforgó ütemezés, idővezérelt ciklikus ütemezés
2. Prioritásos ütemezés

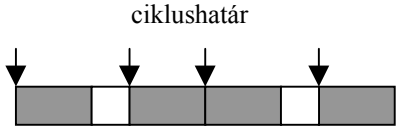
Az alábbiakban ismertetett alapvető SW szervezési módszerek ezen ütemezési technikák megvalósítását adják.

Gyakori vezérlési szerkezetek:

1. Egyszerű ciklikus programszervezés(Round robin)
2. Megszakításokkal kiegészített ciklikus programszervezés (Round robin with interrupts)
3. Ütemezett függvényekkel szervezett program (Function queue scheduling)
4. Valós-idejű operációs rendszerre épülő programszervezés (Preemptive multitasking, microkernel, RTOS)

Egyszerű ciklikus programszervezés

```
void main()
{
while (TRUE)
        {
        if (DeviceANeedsService())
                { ServiceA();}
        if (DeviceBNeedsService())
                { ServiceB();}
        }
}
```



A végtelen ciklus jellegzetes megoldás: egyszerű periódikus ütemezést valósít meg. Egyszerűbb esetben a hurokban levő feladatok nem struktúráltak, tehát nincsenek egységbe zárva, nem tekinthetők task-oknak. Ez a *végtelen ciklus*. Összetettebb esetben a hurokban levő lépések önálló összetett feladatok (taskok). Utóbbi esetben a taskok közötti információcsere minimális.

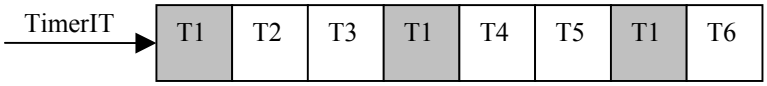
A hardware kezelése: lekérdezéses.

Worst Case kiszolgálás idő: Az összes task futási idejének összege. (a példában: t_A+t_B)

A ciklusok hossza változik a hardware olvasások miatt. Így ugyanazon egység olvasásának gyakorisága is változik. A task-ok végrehajtása a lehető leggyakrabban történik. A processzor kihasználtsága maximális.

A task-ok közötti kommunikáció gond nélkül történhet megosztott változókkal, hiszen mindig csak az egyik fut a feladat befejezéséig.

Továbbfejlesztések:
a.) Különböző gyakorsággal futtatott task-ok.



Módosított formájában a ciklust időzítőáramkör kezdeményezi. Ekkor is minden task futási idejének maximális összege kisebb kell legyen a timer periódusidejénél.

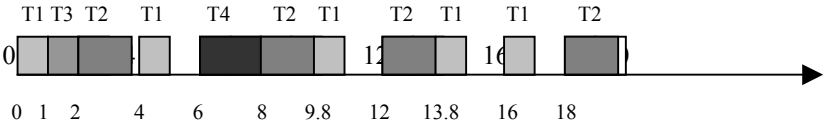
b.) Periódikus idővezérelt ütemezés
Ennek az ütemezésnek a lényege az, hogy az aktuálisan elvégzendő feladatról előre ismert időpontokban születik döntés. (ütemezési pontok). Azon rendszerekben használható, ahol a hard real-time követelmények a-priori ismertek és rögzítettek, a taskok periódikusak, paramétereik ismertek. A döntéseket célszerű rendszeres időközönként meghozni, így a döntési pillanatok időzítőáramkör segítségével kijelölhetők.

Modell:

- állandó számú periodikus task
- a taskok paraméterei a-priori ismertek
- a taskok futásra kész állapota ismert időpontokban történik (ez gyakorlatilag azt jelenti, hogy ismerjük a task-végrehajtás szükséges gyakoriságát. Például egy 1 kHz mintavételező rendszernél a mérési eredményt 1 ms-onként kapjuk, a feldolgozást ilyen gyakran kell végezni.)

Példa:
T1: periódusidő: 4, futási idő (worst case): 1; T2: 5, 1.8; T3: 20, 1; T4: 20, 2. Minden task határideje megegyezik a periódusidejével, fázisa 0.

Ezt a rendszert célszerű egy 20 időtartamú ciklusba szervezni (hypeperiod, major cycle)



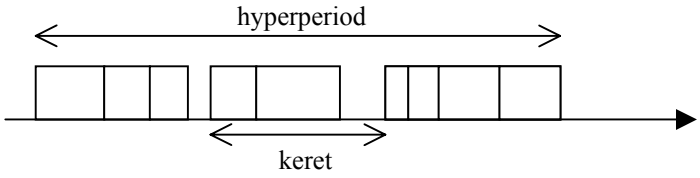
A szabad területeket lehetőség szerint "periódikusan" kell meghagyni. A nem periodikus eseményekre indítandó feladatok megoldhatók ezekben az időszeletekben (ha befejezhető az adott időben).
Az ütemezést egy táblázatban tároljuk:

Időpont (time)	Task függvény referencia (t_func)
0	func_T1
1	func_T3
2	func_T2
4	func_T1
6	func_T4
8	func_T2
9.8	func_T1
12	func_T2
13.8	func_T1
16	func_T1
18	func_T2

```
main()
{ /*H: hyperperiod, N: nr of tasks*/
k=0;i=0;
SetTimer(table(k,time));
while(1)
        {
        while (!TIMER_FLAG);
        TIMER_FLAG=0;
        act_function_poi=table(k,t_func);
        i++;
        k=modulo(I,N);
        SetTimer(H*lower(i/N)+table(k,time));
        act_function_poi();
        }
}
void interrupt TimerIT()
{TIMER_FLAG=1;}
```

A Timer megszakítás futási idejét és az ütemező rutin futási idejét hozzá kell adni a task futási idejéhez.

Szisztematikus tervezés
Az aktuálisan futtatandó task-ok kiválasztását célszerű periódikus időpontokhoz kötni. Ezek a periódusok a *keretek (frame-f*). Egy ilyen rendszerben két ciklus fut: az egyik a *hyperperiodus*, a másik a hyperperioduson belül elhelyezett egész számú *keret* (*frame*). Nyilván a keretek hosszának megválasztása a task-októl függ. A kereten belül nincs kiürítés, a taskok fázisa a keret hosszának egész számú többszöröse.



Minden keret elején célszerű az alábbi vizsgálatokat megtenni:

- A futtatandó task aktívvá (futásra kész) vált-e?
- Történt-e túlfutás?

Ahhoz, hogy ezeket a vizsgálatokat a keretek elején el tudjuk végezni, továbbá, hogy a task-ok worst case határidejükön belül biztosan befejezzék futásukat a keretek megválasztására megkötéseket kell bevezetni:

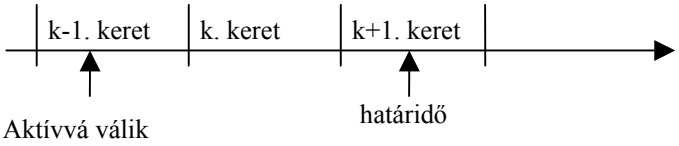
1. A keret hossza (f) elég hosszú legyen ahhoz, hogy a taskok mindegyike be tudja fejezni a munkáját

$$f \geq \max_{1<i<n}(C_i)$$

2. A hiperciklus hosszát a lehető legkisebbre kell választani úgy, az a keret egész számú többszöröse legyen. Ezért legalább egy task periódusidjére (T_i) igaz kell legyen a következő összefüggés:

$$\left\lfloor \frac{T_i}{f} \right\rfloor - \frac{T_i}{f} = 0$$

3. Annak eldöntéséhez, hogy minden feladatot elvégeztünk az előző frame-ben, azaz minden task teljesíti a worst-case határidejét szükséges, hogy a keret olyan kicsi legyen, hogy minden task aktívvá válása és a határideje között legalább egy keret legyen:



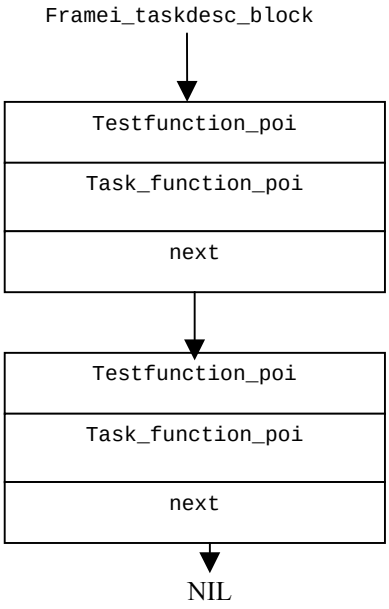
A feltételek együttes teljesítéséhez sokszor egy task-ot apróbb feladatokra kell vágni és azokat különböző keretekbe elhelyezni. A tervezés feladata tehát:

1. A keret méret meghatározása
2. Minden task felbontása függvényekre
3. A függvények elhelyezése a keretekbe

Módosított algoritmus (*Cyclic executive*):

Táblázat:

Frame sorszám	Adott frame-ben végrehajtandó feladatleírók láncolt listája
0	Frame0_taskdesc_block
1	Frame1_taskdesc_block
2	Frame2_taskdesc_block
3	Frame3_taskdesc_block
4	Frame4_taskdesc_block
5	Frame5_taskdesc_block
6	Frame6_taskdesc_block




```
main()
       { /*F frame hossz; t time; k frames*/
       t=0; k=0;
       while (1)
              {
              while (! TIMER_FLAG);
              TIMER_FLAG=0;
              CurrentBlock=table(k);
              t++;
              k=modulo (t,F);
              /*végignézni a blokk összes elemét a
              tesztfüggvényeket meghívni és esetleg
              hibakezelés*/
              /*végigjárni a blokkot és az össze task függvényt
              meghívni egymás után*/
              }
       }
```

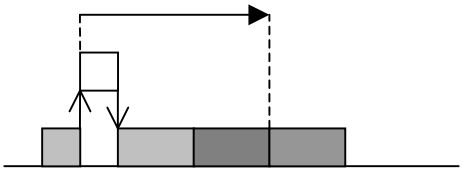
A rendszer inkrementális továbbfejlesztése problematikus: a valós-idejű követelményeknek megfelelő rendszer egy új feladat integrálásával gyakran már nem teljesíti az előírásokat. Mindig újra kell tervezni.
Csak nagyon egyszerű esetben használható. Jól kiszámítható működés, könnyen tesztelhető. Ahol a ciklus elég gyors az események kiszolgálásához ott megfelelő.

Megszakításokkal kiegészített ciklikus programszervezés

```
FLAG FLAG_A, FLAG_B;
void interrupt A_Handler()    {FLAG_A=TRUE;}
void interrupt B_Handler()    {FLAG_B=TRUE;}

void main()  {
       while (TRUE){
              if FLAG_A   {
                 FLAG_A=FALSE; ServiceA();
              }
              if FLAG_B   {
                 FLAG_B=FALSE; ServiceB();
              }
       }
}
```

A rendszer válaszideje az adott eseményre



A hardware kezelése megszakítással történhet, így ez gyorsabb eszközkezelést ad. Az IT rutinban elvégezhetőek azok a feladatok, amelyek a megszakítást küldő periféria gyors kezelését adják. Az előző megoldások csak a timer megszakítást kezelték, az egyéb perifériák elérése lekérdezéssel történt.

Worst-case válaszidő minden egyéb task idejének összege (itt A jelzésre tB). A taskok prioritása azonos, csak a jelzés gyorsul, a kiszolgálás nem. Fő ciklusban ellenőrzés: A,B,A,C,A… Ezzel a válaszidő csökkenthető.

Megszakításokhoz **prioritás** rendelhető. Sokszor a hardware megoldja a prioritást, így a megszakítások egymásba is ágyazhatók, ezzel nem vesztünk el fontos jelzést.

A kódfejlesztésre mutatott **érzékenység**: IT- jó, task - rossz

Megosztott változók problémája jelentkezik task-és IT rutin között. Módosított formában az alapciklust ismét időzítő indíthatja. Minden timer ciklus elején azonban van egy rövid task, amely alatt a megszakítások tiltva vannak és a task az IT változókat a task változókba másolja. A task-ok futásukat be kell fejezzék a következő timer megszakításig. Task-ok közötti **kommunikáció** könnyen megoldható megosztott változókkal.

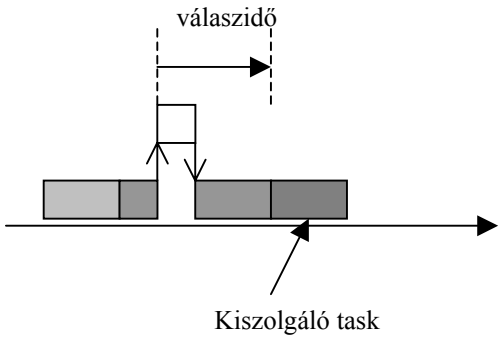
A rendszer **kiszámíthatóságát** a külső események rontják. Ahol az események gyakorisága tervezhető (pl. állandó mintavételi frekvencia, állandó sebességű kommunikációs csatorna) és a taskokra előírt válaszidő nem túl kritikus ott megfelelő megoldás. Ehhez azonban az kell hogy a taskok végrehajtási ideje ne térjen el egymástól jelentősen. (nyomtatás - mérésfeldolgozás)

Ütemezett függvényekkel szervezett program

```
void interrupt A_Handler()
       {  HandleHW_A(); Put_Function(ServiceA);  }
void interrupt B_Handler()
       {  HandleHW_B(); Put_Function(ServiceB);  }

void ServiceA();
void ServiceB();

void main() {
       while (TRUE) {
       while (IsFunctionQueueEmpty());
       CallFirstFromQueue();
       }
}
```



Működés: Az egyes megszakítás-kezelő rutinok függvényreferenciákat helyeznek egy közös queue-ba. A főciklus pedig mindig a queue legfelső elemét hívja meg. A legfelső azt jelenti, hogy adott prioritási rendszerben a legfontosabbat. A megszakítás-kezelő rutinok bizonyos HW feltételek vizsgálatával különböző helyzetekben más-más függvényeket is helyezhetnek a queue-ba.
A módszer előnye az előzőhöz képest, hogy az aktuális task befejezését követően mindig a legmagasabb prioritású következik. Ezzel a válaszidő a leghosszabb task válaszidejére csökken. A megszakításokhoz és a függvényekhez is lehet prioritást rendelni, de ezek többnyire összhangban vannak.

A megszakításokhoz **prioritás** rendelhető
Tetszőleges task prioritás beállítható kiemelési algoritmussal. A kiemelési algoritmus bonyolult is lehet, így a például dinamikus prioritási rendszer is megvalósítható.

Worst Case válaszidő a leghosszabb task ideje.

A módszer maradó hátránya az, hogy mivel nem preemptív, így az éppen futó alacsony prioritású task késlelteti az eseménykiszolgálást. Ez a késleltetés pedig meglehetősen nagy is lehet (pl. nyomtatás).

Érzékenység: kódfejlesztésre viszonylag érzéketlen: újabb események kezelése, újabb megszakítás-kezelő rutinok felvételével és a task a prioritási rendszerbe való beillesztésével könnyen végezhető. A rendszer válaszideje az új task végrehajtási idejétől függ.

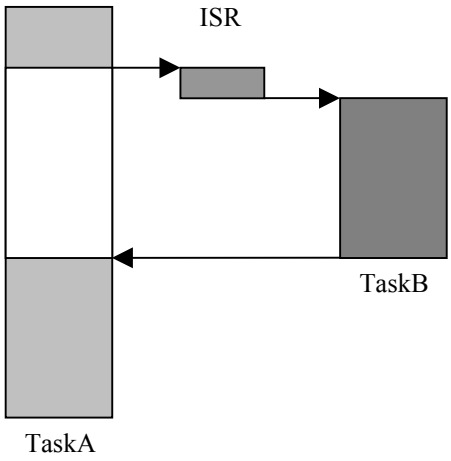
Megosztott változók problémája: itt is jelentkezik a megosztott változók problémája task-megszakítás viszonylatban. A taskok közötti **kommunikáció** biztonságos.

RTOS

(mikrokernel architectúra, valós-idejű operációs rendszerre épített architektúra)

```
void interrupt A_Handler()
       {  HandleHW_A(); SignalX());  }
void interrupt B_Handler()
       {  HandleHW_B(); SignalY());  }

void TaskA(void) {
       while (TRUE) {
       WaitForSignalX(); ServiceA();
       }
}
void TaskB(void) {
       while (TRUE) {
       WaitForSignalY(); ServiceB();
       }
}
```



Működés:
A megszakítás-kezelő rutinok a sürgős hardware feladatok elvégzése után az operációs rendszer segítségével üzennek a megfelelő task-oknak. Az üzenet sokféle lehet, a példában jelzést küldenek. Az egyes task-okhoz prioritások tartoznak. Amennyiben a megszakítás az éppen futó task-nál magasabb prioritású task-nak küldött üzenetet, akkor az ütemező az éppen futó task-ot felfüggeszti és a magasabb prioritású kezd futni. A jelzés megvalósítása és a task-váltás az operációs rendszer feladata.

A módszer preemptív, ezért a nagyon sürgős események kiszolgálása a megszakítás-kezelő rutin után azonnal elkezdődik.

A **worst-case válaszidő** az operációs rendszer jellemző adata: ~ 10 us

Érzékenység: a rendszer nem érzékeny új események bevitelére, jól fejleszthető és karbantartható.

Jelentősen eltérő kiszolgálási idejű feladatok is jól tervezhetők.

Felhasználható minden rendszerben. Az operációs rendszer programjának processzorigénye csökkenti a processzoridőt. A kódméret az operációs rendszer kódjával megnő. Az operációs rendszer által biztosított válaszidő hosszabb, mint a közvetlen hardware-kezelő kód válaszideje. Ezen szempontok, és a módszerrel kínált jó tervezhetőség kompromisszuma dönti el az alkalmazását.