Szenzorhálózatok programfejlesztési kérdései

Orosz György

2011. 09. 30.

Szoftverfejlesztési alternatívák

- Erőforráskorlátok! (CPU, MEM, Energia)
- PC-től eltérő felfogás: HW közeli programozás
- Eszközök közvetlen kezelése:
  - Lekérdezés
  - IT (megszakítás)
- Assembly
  - Kisebb feladatok megoldása
  - Időkritikus alkalmazások
  - Nehézkes (tovább)fejlesztés és debugolás
- Magasszintű programozási nyelv (C, Java???)
  - Kisebb hatásfok (nem feltétlenül)
  - Gyorsabb fejlesztés
  - Továbbfejlesztés egyszerűbb, skálázhatóság
- Beágyazott operációs rendszer

Feladatkiszolgálás

- Alapvető feladat: események kezelése
  - Kommunikáció (rádió/protokoll kezelés)
  - Megfigyelések végzése
  - Perifériák kiszolgálása
  - Időzítések
- Processzor: szekvenciális végrehajtás
- A feladatok kezelését tervezni kell (ütemezés)
- Szempontok
  - Feladat kiszolgálásának hossza (végrehajtási ideje)
  - Feladat fontossága (válaszidő biztosítása)
  - Megfelelő processzorkihasználtság: jól el kell osztani a feladatokat
  - Példa: egy gyorsan változó mennyiség (pl. hang, rezgés) mérését pontosan kell elvégezni, itt nem engedhető meg változó válaszidő. Az adatok továbbításának ütemezése viszont nem kritikus, lényeg, hogy elküldjük valamikor az adatokat: elegendő kisebb prioritás.
- Szenzorhálózat: elosztott rendszer.
  - Ugyanaz a program fut sok eszközön
  - Gyakorlat kell annak átlátásához, hogy egyazon program hogyan működik több eszközön eltérő üzemállapotban, eltérő körülmények között

Feladatok kiszolgálása (ütemezés) Példák

- Egyszerű ciklikus programszervezés
  - Események egymás utáni lekérdezése és kiszolgálása

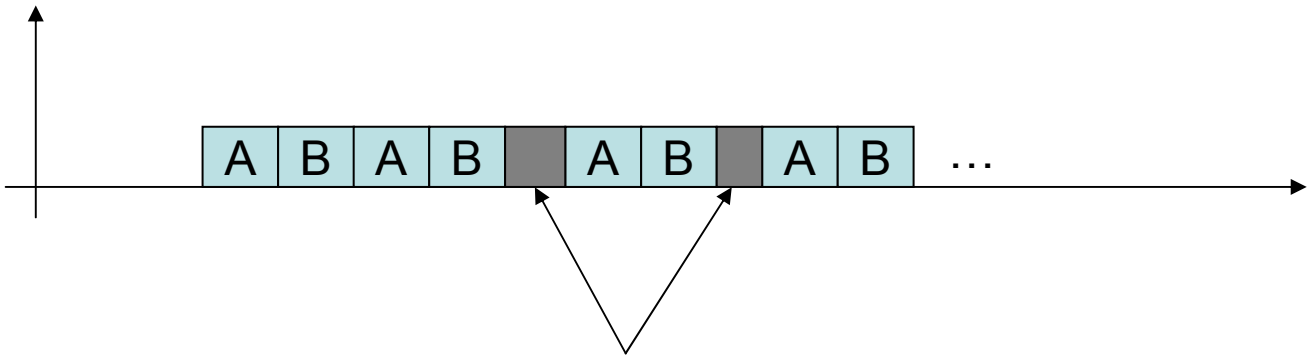
```
while(true){
        if (eventARequest==true) {serviceEventA();}
        if (eventBRequest==true) {serviceEventB();}
}
```

- Folyamatos lekérdezés → maximális CPU kihasználtság (fogyasztás ☹)
- Worst case válaszidő: $T_A+T_B+\ldots$: garantált (kérés kezdete-kiszolgálás vége)
- Gyorsan fejleszthető, kiszámítható
- Új feladatok hozzáadásával nő a válaszidő
- Nem preemptív
  - kölcsönös kizárás nem gond (nem fut egyszerre két folyamat) ☺
  - Kis válaszidejű események kiszolgálása nem garantált ☹
- „Súlyozás” is lehet, pl.: ABAAC ABAAC …(‘A’ többször fut)
- HW kezelés: lekérdezéssel

Egyszerű ciklikus programszervezés

- Példák:
  - adat lekérdezése (A) és továbbítása (B)
  - rádió folyamatos lekérdezése (A), ha van üzenet annak feldolgozása (B)

- Időzítési diagram:

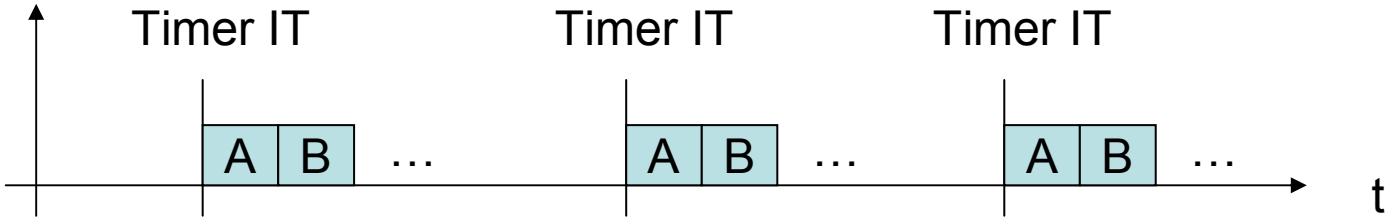


folytonos lekérdezés végzése: nem kell kiszolgálni egyik taszkot sem

Ütemezés

- Időzített periodikus ütemezés: bizonyos időközönként hajtunk végre egy feladatszekvenciát

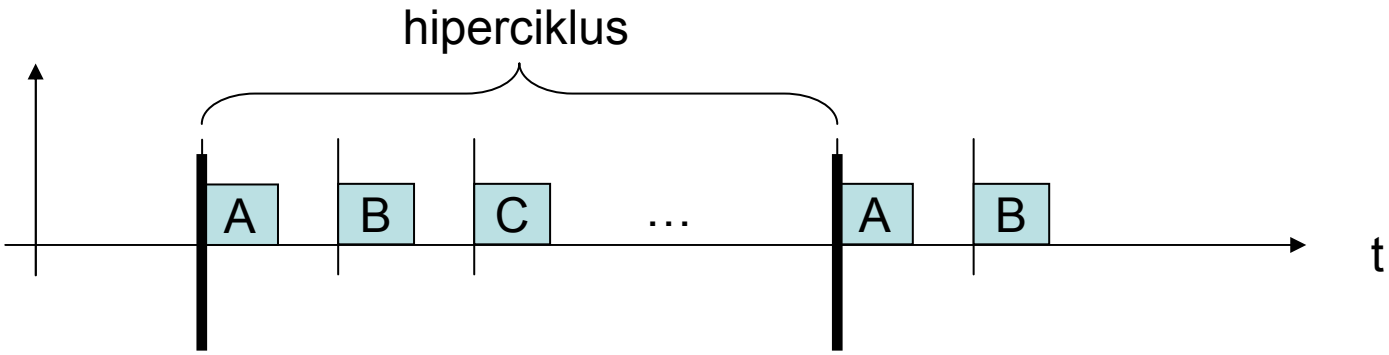
```
TimerITServiceRoutine(){
        if (eventARequest==true) {serviceEventA();}
        if (eventBRequest==true) {serviceEventB();}
}
```



- Egyszerű ciklikus programszervezéshez hasonló tulajdonságok, de csak adott időközönként történik kiszolgálás
- Itt is lehet „súlyozni” (IT:AAABAB … IT: AAABAB …)
- Jobb energiafelhasználás: van idle szakasz

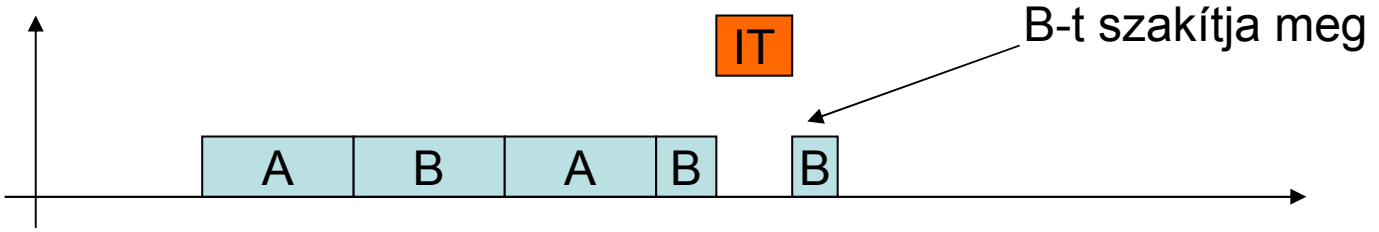
Idővezérelt ütemezés 2

- Minden feladathoz saját időpont (adminisztráció)



- Kiszámítható működés
- Energiafelhasználás ☺
- Kölcsönös kizárás ☺ (nincsenek megszakítások)
- Fejlesztés nehézkes: időzítések újratervezése

Megszakításokkal (IT) való kiegészítés



- IT (interrupt) kell: lekezelés nem lehet lekérdezéssel (időkritikus)
- A determinisztikusságot elrontja
- Kölcsönös kizárást biztosítani az IT-kre
- Válaszidő megnő az IT-k idejével
- Elterjedt megoldás (általában sok feladat megkívánja)
- Taszkok hozzáadásával nő a válaszidő
- IT rutin: fontos feladatok, feldolgozás később

Ütemezett függvények

- IT vezérelt
- IT során egy várósorba (queue) tesszük a végrehajtandó taszkokat (fv.-ket), innen valamilyen sorrendben végrehajtjuk
- TinyOS is hasonló szisztémát követ: putQueue → post task

```
ITServiceRoutine_A(){
        fastHWHandleA(); putQueue(serviceA);
}
ITServiceRoutine_B(){
        fastHWHandleB(); putQueue(serviceB);
}
.
main(){
    while(1){  //gond: processzor folyton fut
        if (queueIsNotEmpty()){
             queueCallNextItem();
        }
    }
}
```

Ütemezett függvények

- Válaszidő: leghosszabb folyamat
- Kölcsönös kizárásra figyelni IT-k esetén
- Egyszerű fejleszthetőség
  - Függvények hozzáadása a sorhoz
  - A függvényeket nem kell megkülönböztetni
- A queue-ból való kiemelés prioritásos alapon is mehet
  - Fontos feladatok válaszideje csökken
- Nem preemptív

Operációs rendszer (OS)

- A programozót tehermentesíti az ütemezési feladatok implementálása alól (folyamatok tervezése marad)
- Biztosítja a párhuzamos programozáshoz szükséges környezetet
- Transzparens HW kezelést biztosít
  - Pl. analóg-digitális átalakítás, időzítő …
  - API: Application Programming Interface
  - Platformfüggetlen (cél)
- Szinkronizációs, kommunikációs primitívek: pl. szemafor, mutex, queue …
- Memóriakezelés

Operációs rendszer

- Fejleszthetőség jó
- Extra erőforrások szükségesek a futásához ☹
  - Futási idő, pl. context switching
  - Memória: adminisztráció, alapkomponensek
- Taszkok közötti kommunikáció OS szinten
- Processzor kihasználás/energia ☺ (idle)
- Prioritás kezelhető
- Preemptív / nem preemptív: OS függő
- Gyakran extra funkcionalitások (operációs rendszer függő), például:
  - TinyOS: rádiókezelés, magasabb szintű protokollok
  - Filekezelés
  - TCP/IP kommunikáció

Hagyományos vs. Beágyazott operációs rendszerek

	PC	beágyazott
Indulás	Bootloader: OS-t indít, aztán alkalmazások	Reset→ alkalmazás → OS
Memóriavédelem	Erős védelem	Gyenge védelem (erőforrás)
Kód felépítés	Külön OS, külön kód	Egyben fordul a kettő
Válaszidő	Nem determinisztikus. Több új folyamat, dinamikus felépítés.	Számítható
Kódhatékonyság	Nagy tartalék az eszközökben	Csak a szükséges kódrészek