Beágyazott információs rendszerek

Valós idejű kommunikáció

Valós idejű operációs rendszerek

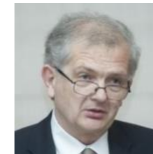
2020. november 18.

Időszinkronizáció vezetéknélküli hálózatokban

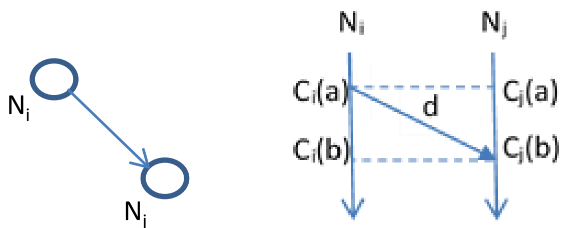
Emlékeztető

A szinkronizáció osztályai/fajtái:

- *Külső, belső* (pontosság, együttfutás);
- *Időtartam szerint*: (1) folytonos (mindig fennáll), (2) kérésre (on demand);
- *Eseményvezérelt:* az időbélyeg előállításához csak akkor kell szinkronizált idő, amikor az esemény bekövetkezett (Post-facto synchronization);
- *Idővezérelt:* akkor használjuk, amikor több szenzor adatára van szükségünk egy adott időpontra;
  (1) közvetlen (immediate): Vegyen mintát azonnal, és rendeljen hozzá időbélyeget;
  (2) Előre megadott (anticipated) időpontbeli mintavétel, amit akkor használunk, ha az adattovábbítás párhuzamos megvalósítása nehézségekbe ütközik, vagy ha több ugrás (hop) szükséges. (A megfelelő minőségű szinkronizált állapotot elérése csak az előre megadott időpontra esedékes.) (Pre-facto synchronization);
- *Érintettség* (Scope): minden csomópont vagy csak egy részhalmaz:
  (1) hálózati topológia szerinti;
  (2) időbeni érintettség.
- *Ütem, ofszet szinkronizáció*: (1) Ütem: a csomópontok azonos időintervallum-hosszúságot mérnek (pl. egy objektum feltűnésének és eltűnésének időpontjai között eltelt idő; (2) Ofszet: minden érintett csomópont azonos időt mutat.
- *Időskála szinkronizáció*: helyi idő transzformálása egy másik csomópont helyi idejébe.
- *Óra szinkrozináció*: ütem és ofszet szinkronizáció vagy folyamatos szinkronizáció.
- *Időpillanatok szerint*: idő+bizonytalanság (valószínűség eloszlás): $t = 5$ + bizonytalanság
- *Időintervallumok szerint*: $t \in [4.5, 5.5]$, nagyon jó, ha garantálni lehet.



Egyirányú szinkronizáció:



Az N_j csomópont nem ismeri d-t, csak azt, hogy az N_i csomópont órája $C_i(a)$ értéket azt megelőzően mutatott, hogy az N_j csomópont órája $C_j(b)$-t.

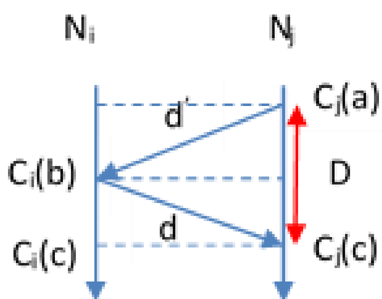
Ahhoz, hogy szinkronizálni tudjunk vagy $C_j(a)$ vagy $C_i(b)$ értékét meg kell becsüljük.

Ha ismert $d_{min} \leq d \leq d_{max}$, akkor

$$\hat{C}_j(a) \approx C_j(b) - \frac{d_{min} + d_{max}}{2}$$ vagy $$\hat{C}_i(b) \approx C_i(a) + \frac{d_{min} + d_{max}}{2}$$

Ezek ismeretében az N_j csomópont óráját vagy $\hat{C}_j(a) - C_i(a)$ értékkel vagy $C_j(b) - \hat{C}_i(b)$ értékkel kell késleltetnünk/visszaállítanunk. Ha a kommunikáció jittere $(d_{max} - d_{min})$ nagy, akkor az így végrehajtott szinkronizáció pontatlan lesz, hiszen például $C_j(a)$ alsó határa $C_j(b) - d_{max}$, felső határa pedig $C_j(b) - d_{min}$ értékkel adható meg, ami ilyenkor széles tartomány.

Kétirányú (oda-vissza, round trip) szinkronizáció:

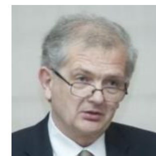


Az N_j csomópont tudja, hogy $0 \leq d \leq D$. $D = C_j(c) - C_j(a)$.

Ha $d_{min} \leq d \leq d_{max}$, akkor $\max(D - d_{max}, d_{min})$ és $\min(d_{max}, D - d_{min})$ adják *d* korlátait.

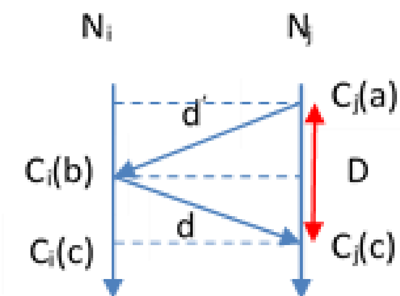
Az itt számítható becslő: $\hat{C}_j(b) \approx C_j(c) - \frac{D}{2}$,

aminek alsó határa $C_j(c) - (D - d_{min})$, felső határa pedig $C_j(c) - d_{min}$.

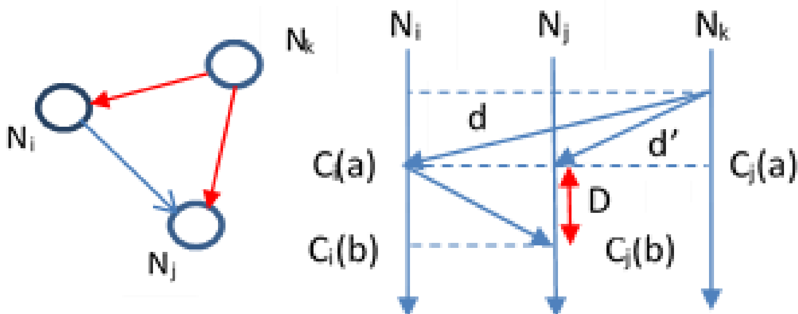


Itt az N_j csomópont óráját a $\hat{C}_j(b) - C_i(b)$ értékkel kell késleltetni/visszaállítani.

Ezzel a megoldással jobb minőségű szinkronizáció érhető el.
A worst-case szinkronizációs hiba: $\frac{D}{2} - d_{min}$, ami az ábra alapján is könnyen belátható. A módszer pontossága javítható az ún. ***valószínűségi időszinkronizációval***, amely esetében a vétel után az N_j csomópont ellenőrzi, hogy a $\frac{D}{2} - d_{min} <$ **egy specifikált küszöbnél.** Ha nem, akkor ismétel!



Anoním (Reference broadcasting) szinkronizáció: Az egyirányú és a kétirányú előnyeit ötvözi.



A szinkronizáció kezdeményezője az N_k csomópont. Azt használjuk ki, hogy miközben a kommunikáció ideje változó, a broadcasting jellegből adódóan $d \approx d'$. Ezzel $\hat{C}_i(b) \approx C_i(a) + D$, aminek ismeretében az N_j csomópont óráját $C_j(b) - \hat{C}_i(b)$ értékkel kell késleltetnünk/visszaállítanunk. Fontos sajátosság, hogy az N_j csomópont szinkronizálása rádió adójának használata nélkül valósul meg.

Több csomópont szinkronizálása:

(1) egy-ugrásos, több master, a masterek szinkronizálása kívülről (pl. GPS);
(2) klaszterek felállítása, köztük idő-gateway-ek;
(3) fa struktúra: master a gyökérben: több-ugrásos. Távolodva romlik a pontosság.

Beágyazott operációs rendszerek

Bevezetés: **Beágyazott rendszerek szoftver vonatkozásai, tipikus szoftver architektúrák**

Szempontok: számítási kapacitás, memóriaméret (RAM, ROM), fejleszthetőség, **továbbfejleszthetőség**, **reakcióidő** külső, aszinkron esemény esetén, **védelem** (memória), rekurzió, függvények **újrahívásának** támogatása, processzor kihasználtsága.

Tulajdonságok, amik alapján minősítjük az egyes megoldásokat:

- maximális válaszidő,
- hardverkezelés megvalósítása,
- taszkok közötti kommunikáció megvalósítása,
- tervezhetőség,
- alkalmazási kör.

Gyakorlati megvalósítás szempontjából:

- ciklikus programszervezés,
- IT-vel kiegészített ciklikus programszervezés
- ütemezett függvények módszere
- **RTOS**

Egyszerű ciklikus programszervezés:

A processzor végtelen ciklusban pörög, akkor is fut, amikor senki sem igényel kiszolgálást.

Szoftver architektúrák osztályozása:

- periodikus,
- prioritásos,
- eseményvezérelt,
- idővezérelt.

Ciklikus programszervezés

- körforgó
- súlyozott körforgó
- idővezérelt körforgó
- szigorúan idővezérelt

```
void main() {
while (TRUE){
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceB_Needs_Service()) {Service_B};
if (DeviceC_Needs_Service()) {Service_C};
...
}
}
```

Egyszerű ciklikus programszervezés:

Tulajdonságok:

- maximális válaszidő: tA+ tB+ tC+ ..., azaz a **maximális ciklusidő**
- hardverkezelés: **lekérdezéssel** (polling)
- taszkok közötti kommunikáció: **megosztott változókkal** (nem preemptív, így nem gond!)
- fejleszthetőség: **rossz**
- HRT viselkedés: **lassú** (pl. nyomtatási taszk) (ettől még lehet RT)
- processzor kihasználtság:**100%** (ez NEM jó!)
- alkalmazási kör: ahol a rendszer időállandója nagyobb a ciklus futásánál (**gyors és ritka események**)

Súlyozott körforgó programszervezés:

a gyakoribb taszkok a cikluson belül ismétlődhetnek.

Tulajdonságok:

- max. válaszidő: tA+ tB+ tA+ tC+ tA+.., de gyakoribb taszkokra **kisebb,** mint a maximális ciklusidő
- hardverkezelés: **lekérdezéssel** (polling)
- taszkok közötti kommunikáció: **megosztott változókkal** (nem preemptív, így nem gond!)
- fejleszthetőség: **rossz**
- processzor kihasználtság: továbbra is **100%**
- egyéb tulajdonság: **prioritás jellegű viselkedés**, de NEM preemptív

```
void main() {
while (TRUE){
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceB_Needs_Service()) {Service_B};
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceC_Needs_Service()) {Service_C};
if (DeviceA_Needs_Service()) {Service_A};
...
}
}
```

Idővezérelt körforgó programszervezés:

A ciklus határokat egy **timer adja** (csak a határokat!). Timer IT-nként egyszer vagy többször lefut a ciklus. Egy cikluson belül lehet súlyozott körforgó.

Tulajdonságok:

- max. válaszidő: **ciklus periódusideje**
- hardverkezelés: **lekérdezéssel** (polling)
- taszkok közötti kommunikáció: **megosztott változókkal**
- fejleszthetőség: **rossz**
- processzor kihasználtság**:<100%,** van standby

Szigorúan idővezérelt programszervezés (time-triggered protokoll):

Minden taszk futása előre meghatározott időben indul.

Adminisztrálás: egy táblázatban az idők és a függvény referenciák (hiper-ciklusonként), egy **mikro futtatórendszer** figyeli az időket, és indítja a „taszkokat”.

Tulajdonságok:

- max. válaszidő: adott taszk ütemezett gyakorisága (+ a taszk futási ideje)
- hardverkezelés: **lekérdezéssel** (polling)
- taszkok közötti kommunikáció: **megosztott változókkal**
- fejleszthetőség: **rossz**
- processzor kihasználtság**:<100%,** van standby
- HRT viselkedés: **OK**, alkalmazása **biztonságkritikus rendszerekben** tipikus

IT-vel kiegészített ciklikus programszervezés:

nem lekérdezéssel, hanem megszakítással jelzünk.

Tulajdonságok:

- max. válaszidő: tA+ tB+ tC+ ...(+ IT) **csak a jelzés gyorsul**, a kiszolgálás nem
- hardverkezelés: **megszakítással**, prioritás lehetősége
- taszkok közötti kommunikáció: **megosztott változókkal**. Taszk-taszk között: **nem gond.** IT-taszk között: **megosztott változók problémája**
- fejleszthetőség: **IT szempontjából jó**, de taszkok hozzáadásával megváltoznak a viszonyok

```
FLAG A, B, C;
void interrupt A_Handler() { Handle_HW_A(); A=TRUE; }
void interrupt B_Handler() { Handle_HW_B(); B=TRUE; }
void interrupt C_Handler() { Handle_HW_C(); C=TRUE; }
void main() {
while (TRUE){
 if A {A=FALSE; Service_A(); }
 if B { B=FALSE; Service_B(); }
 if C { C=FALSE; Service_C(); }
...
}
}
```

- alkalmazási kör: ha a taszkok futási ideje **kb. azonos. Ez a legelterjedtebb.**

Ütemezett függvények módszere

Tulajdonságok:

- max. válaszidő: leghosszabb taszk futási ideje+ i. taszk futási ideje
- hardverkezelés: **megszakítással**
- taszkok közötti kommunikáció: **megosztott változókkal**. Taszk-taszk között: **nem gond.** IT-taszk között: **megosztott változók problémája**

```
void interrupt A_Handler() { Handle_HW_A();
PutFunction(Service_A); }
void interrupt B_Handler() { Handle_HW_B();
PutFunction(Service_B); }
void interrupt C_Handler() { Handle_HW_C();
PutFunction(Service_C); }
void Service_A();void Service_B();void Service_C();
void main() {
        while (TRUE){
                while (IsFunctionQueueEmpty());
                        CallFirstFromQueue();
        }
}
```

Ütemezett függvények módszere

- queue-ból való kiemelés sorrendje lehet: (1) **FIFO**, (2) **prioritás alapján**
- hátrány: továbbra sem preemptív
- processzor kihasználtság:**100%**

Valósidejű operációs rendszerre épített szoftver

```
void interrupt A_Handler() { Handle_HW_A(); Signal_A(); }
void interrupt B_Handler() { Handle_HW_B(); Signal_B(); }
void Service_A();
void Service_B();
void task_A(void) {
	while (TRUE){
		Wait_for_Signal_A(); Service_A();
	}
}
void task_B(void) {
	while (TRUE){
		Wait_for_Signal_B(); Service_B();
	}
}
```

Tulajdonságok:

- max. válaszidő: op. rendszer. jellemző adata (~10 usec) + a taszk futási ideje. Ebben figyelembe vesszük a magasabb prioritású taszkok futási idejét is.
- hardverkezelés: **megszakítással**
- taszkok közötti kommunikáció: RTOS kommunikációs függvényekkel. Ez egyben **szinkronizáció** is.
- fejleszthetőség: **nagyon jó**
- HRT viselkedés: **jó**

- processzor kihasználtság:<100%. Mikor? Ha idle alatt sleep!
- alkalmazási kör: **bárhol alkalmazható**
- **hátrány**: operációs rendszer plusz kódot és időt jelent

Alapfogalmak, tulajdonságok, elvárások:

beágyazott OS: kis erőforrásigény (μC-en is elfut)
valósidejű OS: külső eseményre adott véges, determinisztikus válaszidő
taszk: összefüggő tevékenységek sorozata
job: taszkok részfeladatai
process: ütemezési egység, saját memóriaterülete van (taszkokat így implementáljuk)
thread: ütemezési egység, nincs saját memóriája
kernel: OS magja
skálázhatóság: OS szolgáltatásai fordítási időben ki/bekapcsolhatók

Kernel feladatai:

- párhuzamos programozói környezet biztosítása,
- ütemezés,
- taszkok közötti kommunikáció biztosítása,
- megszakítások kezelése
- időzítés,
- memória kezelés

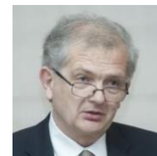
Skálázással bejöhet még:

- perifériák kezelése, rendszerprogramok (API)
- kommunikációs csatornák kezelése
- virtuális memória management, file rendszer stb.

Asztali és beágyazott operációs rendszerek összehasonlítása

a. Az **asztali operációs rendszerek** **nem alkalmasak** beágyazott rendszerekhez, mert:

- szolgáltatásai feleslegesen széleskörűek;
- nem modulárisak, nem hibatűrők, nem konfigurálhatóak, nem módosíthatóak;
- túl nagy tárigényűek;
- energiafogyasztásra nem optimalizáltak;
- nem küldetés-kritikus alkalmazásokra tervezték őket;
- az időzítési bizonytalanságok túl nagyok.



b. Szükség van **konfigurálhatóság**ra:

- egyetlen RTOS nem elégít ki minden igényt;
- a fel nem használt funkciók/adatok okozta overhead nem tolerálható;
- sok olyan beágyazott rendszer van, amelynek nincsen diszkje, billentyűzete, képernyője, egere.

A **konfigurálás** tipikus **eszközei**:

- a felesleges funkciók eltávolítása (például **linker** segítségével);
- feltételes fordítás alkalmazásával (#if és #ifdef parancsok);

Megjegyzés: A verifikáció nehézkes olyan rendszerekben, amelyek nagy számban tartalmaznak konfigurálással származtatott operációs rendszereket:

- minden konfigurálással származtatott operációs rendszert alaposan tesztelni kell;
- pl. az eCos (a Red Hat open source RT operációs rendszere) 100 és 200 közötti konfigurációs ponttal rendelkezik.

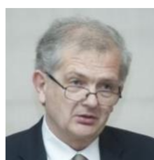
c. A beágyazott operációs rendszerek **eszközmeghajtóit a taszkok kezelik**, nem pedig integrált meghajtók:

- a jósolhatóságot javítja, ha mindent az ütemező kezel;
- praktikusan nincs olyan eszköz, amelyet az operációs rendszer minden változata támogatna, legfeljebb a rendszer időzítő.

Beágyazott RTOS	Standard OS
alkalmazói szoftver	alkalmazói szoftver
middleware-ek	middleware-ek
eszközmeghajtók	operációs rendszer
real-time kernel	eszközmeghajtók

d. A beágyazott operációs rendszerekben **megszakítást bármely taszk használhat**:

- A standard OS-ben súlyos megbízhatatlansági forrás lenne;
- A beágyazott programokról feltételezzük, hogy teszteltek;



- Megengedhető, hogy megszakítás közvetlenül indítson vagy megállítson taszkokat (azáltal, hogy a megszakítás táblázatba a taszkok kezdőcímeit írjuk). Ez hatékonyabb és jósolhatóbb, mint OS szolgáltatásokon keresztül.

Megjegyzés:

- Azonban a **komponálhatóság sérül**: ha egy taszk futását egy megszakításhoz kötjük, akkor nehéz lehet egy másik taszk hozzáadása, amelyet ugyanahhoz az eseményhez kötve kell elindítani.
- Ha a valós-idejű feldolgozás szempont, akkor a megszakítások kiszolgálási idejét figyelembe kell venni. Ebben az esetben a megszakításokat is az ütemezőnek kell kezelnie.

e. A beágyazott operációs rendszerekben **védelmi mechanizmusok nem szükségesek** minden esetben:

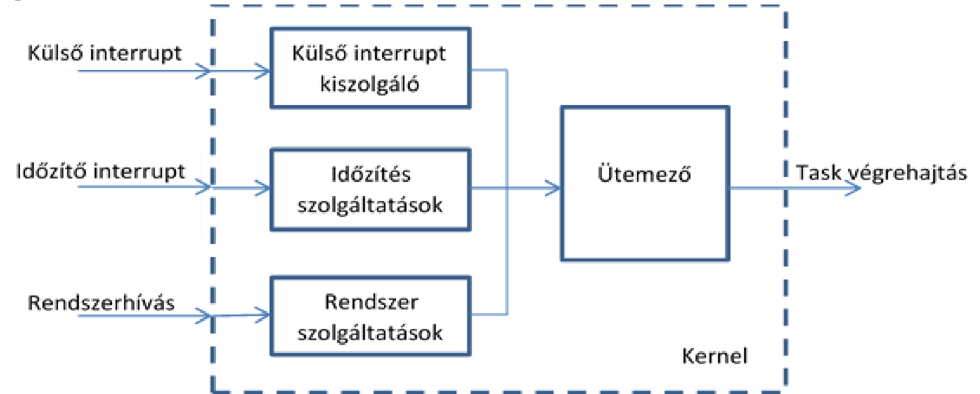
- A beágyazott rendszereket tipikusan egy adott célra tervezik, teszteletlen programot ritkán futtatnak, a szoftvert megbízhatónak tekintik.
- Nincs szükség privilégizált I/O utasításokra, a task-ok el tudják intézni a rájuk vonatkozó I/O műveleteket;
- Természetesen biztonsági szempontok védelmi mechanizmusokat szükségessé tehetnek.

f. A **valós idejű operációs rendszerek (RTOS)** valós idejű rendszerek létrehozását támogatják.

Követelmény:

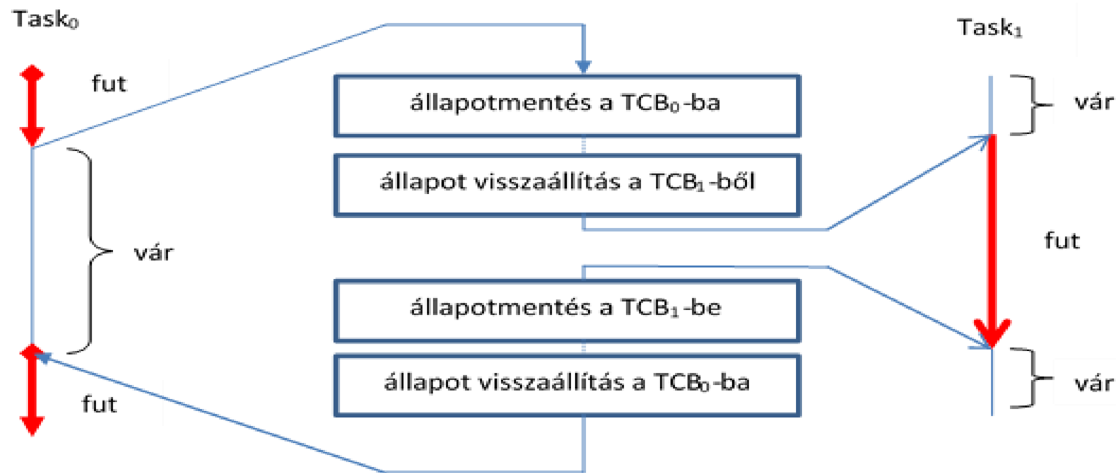
- Az időbeni viselkedés jósolható: minden operációs rendszer szolgáltatás esetében a végrehajtási idő maximuma ismert kell legyen. Az RTOS determinisztikus viselkedésű, majdnem minden tevékenységet az ütemező felügyel.
- Az RTOS intézi az időzítést és az ütemezést: ennek érdekében jó, ha ismeri a taszk-ok határidejeit, és célszerűen nagy felbontású időzítő szolgáltatások biztosít.

- Az RTOS legyen gyors (praktikus megfontolásból).
- Az RTOS folyamat-menedzsment szolgáltatásai:



A kernel feladata:

- A konkurens (kvázi-parallel) feladatok végrehajtása taszkok vagy szálak (threads) formájában:
  - a taszk állapotok kézbentartásával és a taszkok sorba állításával,
  - a taszk preempciók végrehajtásával (gyors context switching) és gyors IT kezeléssel,



- A CPU ütemezése (a határidők garantálása, a task várakozások minimalizálása, a számítási teljesítmény méltányos szétosztása);
- A taszkok szinkronizálása (kritikus szakaszok, szemaforok, monitorok, kölcsönös kizárás);
- A taszkok közötti kommunikáció (bufferelés);
- A valós-idejű óra belső referenciaként történő támogatása.

g. Standard operációs rendszerek valós-idejű kiterjesztései

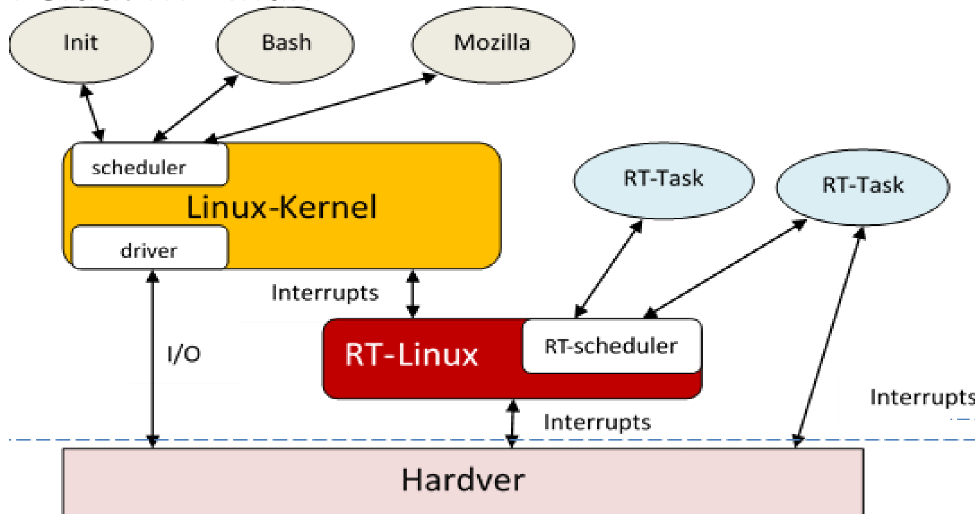
A real-time kernel futtat minden RT taszkot, a standard OS pedig egyetlen taszkként hajtódik végre:

RT-taszk 1	RT-taszk 2	nem-RT taszk 1	nem-RT taszk 2
eszközmeghajtó	eszközmeghajtó	Standard-OS	
real-time kernel			

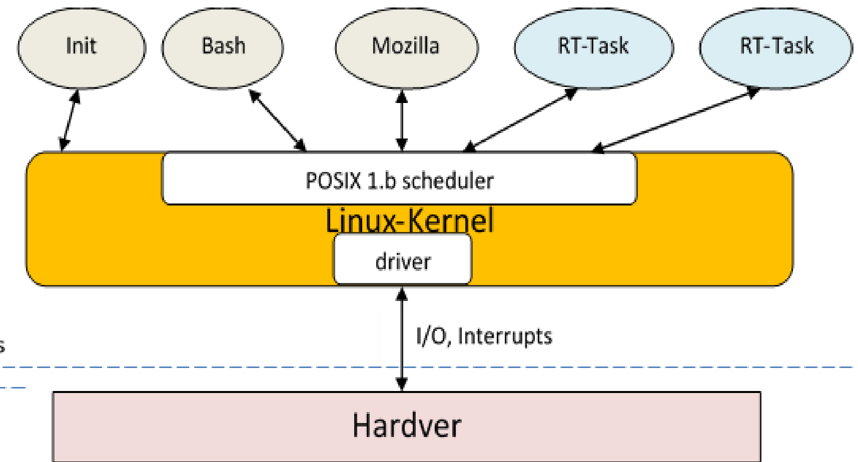
Megjegyzések:

- A standard OS összeomlása nem befolyásolja az RT-taszkok futását;
- Az RT-taszk nem tudja használni a standard OS szolgáltatásokat: a várakozásokat alulmúló elrendezés.

Példa: RT Linux



Példa: Posix 1.b RT-extensions to Linux



A szokványos Linux scheduler lecserélhető a POSIX scheduler-re, amely RT task-ok számára prioritást biztosít. A standard OS hívások mellett speciális RT hívások is vannak.
A programozhatóság egyszerű, de nincs garancia a határidők teljesülésére.

(POSIX: "**P**ortable **O**perating **S**ystem **I**nterface for uni**X**".)

Virtualizáció beágyazott rendszerekben

A szoftver hordozhatóságot szolgálja, azaz fusson különféle hardvereken.

A **virtuális gép** (VM: Virtual Machine) olyan szoftver környezetet biztosít az adott szoftver számára, mintha tényleges hardveren futna az alábbi struktúrában:

Alkalmazás
Operációs rendszer
Hypervisor
Processzor

Az a szoftver réteg, amelyik a virtuális környezetet biztosítja az ún. **virtuális gép monitor** (VMM) vagy **hypervisor.**

Három fő funkciója van:

- az eredeti géppel azonos szoftver környezetet biztosít;
- legfeljebb lassabb a futása;
- teljes mértékben felügyeli a rendszer erőforrásait.

A VM utasítások többsége közvetlenül végrehajtható a hardveren, egy részük **interpreter**-rel valósul meg. Ezek között vannak:

- a vezérlés-érzékeny utasítások, amelyek módosítják a privilegizált gép-állapotokat, ezért interferálnak a hypervisor erőforrások feletti felügyeletével.
- a viselkedés-érzékeny utasítások, amelyek hozzáférnek (olvassák) a privilegizált gép-állapotokat.

Konkurens operációs rendszerek virtuális gépen

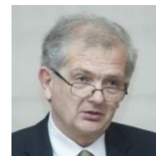
User Interface Software	Access Software
Standard OS	RTOS
Hypervisor	
Processor	

A biztonság növelése virtualizáció alkalmazásával

User Interface Software	Access Software
↓ OS ↑	
Buffer overflow	
OS	
Processor	

User Interface Software	Access Software
↓ OS ↑	OS
Buffer overflow	
Hypervisor	
Processor	

Az egyik alkalmazás okozta hiba nem terjed át a másik alkalmazásra, mert annak saját operációs rendszere van.



Licensz elválasztás virtualizáció alkalmazásával

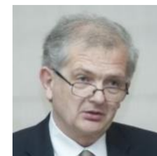
User Interface Software			Access Software		
Linux GPL			RTOS		
	Stub (csonk)			Driver	
Hypervisor					
Processor					

GPL: **General Public Licence.**
A Linux ezen licensz szabályai szerint férhető hozzá szabadon. Ezért minden hozzá készített szoftver is szabad hozzáférésű.

Ahol ez probléma, ott a Linux és célalkalmazás külön virtuális gépeken fut, csak egy ún. csonkot (vagy proxy-t) készítenek, amely hyperhívásokon keresztül éri el a nem szabad hozzáférésű drivert, és azon keresztül a kapcsolódó hardvert.

A virtualizáció korlátai beágyazott rendszerekben:

- Az alkalmazások egyre növekvő komplexitása mellett a több operációs-rendszer futtatás nagyon nagyméretű kódot eredményez, ami önmagában hibaforrás lehet, nagy memóriát igényel, többet fogyaszt.
- Az egyes alrendszerek szoros együttműködése szükséges, ehhez nem passzol a szeparáltság.
- Az egyes alrendszerek hatékony kommunikációja igény, amit a virtuális gép modell nem támogat.
- Az egyes alrendszerek közös erőforrásokon osztoznak, amit nehézkes megszervezni, ha több operációs rendszer fut párhuzamosan.
- A virtualizáció következménye, hogy az ütemezés két szinten történik: (1) Hypervisor és a VM között, (2) minden VM-en futó operációs rendszeren belül.
- A kritikus biztonsági követelmények teljesülését a virtuálizáció egymaga nem támogatja. A kritikus kódrészeket (ún. **trusted computing base**, TCB) privilegizált módban kell futtatni a processzoron. A hypervisor is része a TCB-nek. Az ilyen kódnak bizonyítottan helyesnek kell lennie. A virtualizáció növeli az ilyen kód méretét.



Milyen támogató szoftverre van szükségük a beágyazott rendszereknek?

- támogassa a virtualizáció minden előnyét;
- támogassa az erősen kölcsönhatásban lévő, közepes komplexitású komponensek erős egymásba-ágyazását annak érdekében, hogy a hibás állapotból helyreállni képes, robusztus rendszereket hozzunk létre;
- támogassa a nagy sávszélességű, kis késleltetésű kommunikációt, amely konfigurálható, rendszerszintű biztonsági politikával párosul;
- globális ütemezési stratégia érvényesül a különféle alrendszerek taszkjaira;
- lehessen úgy alrendszereket létrehozni, hogy nagyon kicsi a TCB-jük.

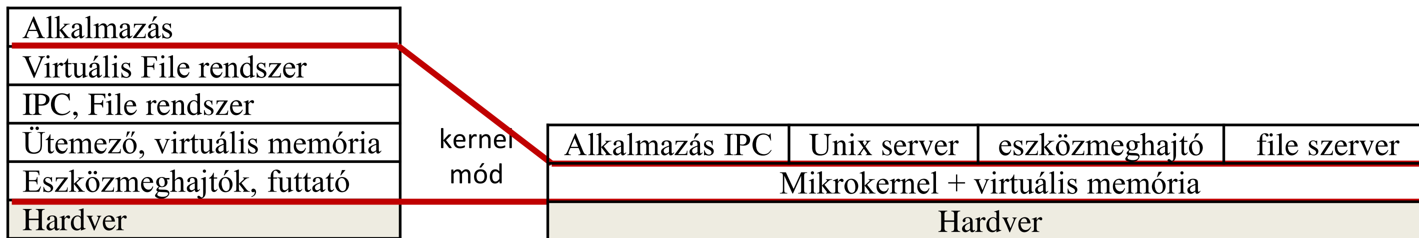
A mikrokernel (microvisor) technológia: a beágyazott rendszerekhez jobban illeszkedő virtualizációs technológia

A **mikrokernel** (**microvisor**) egy minimális privilegizált szoftver réteg, amely csak általános mechanizmusokat biztosít. Az aktuális rendszerszolgáltatások és stratégiák a felhasználói módban futó komponenseken valósulnak meg.

A mikrokernel elve: Egy mikrokernelen belül csak olyan koncepciónak van létjogosultsága, amelyet ha kiviszünk a kernelből, és ezáltal versengő implementációkat engedünk meg, az a megkívánt rendszer funkcionalitás megvalósulását megakadályozná.

A mikrokernel nem nyújt semmilyen szolgáltatást, csak **mechanizmusokat biztosít** szolgáltatások implementálásához.

A hagyományos (monolitikus) operációs rendszer és a mikrokernel alapú rendszerek struktúrájának eltérését a következő ábra mutatja:



A hagyományos struktúra **vertikális jellegű**, a mikrokernel pedig **horizontális**.
Az utóbbinál nincs érdemi különbség az alkalmazás és a rendszerszolgáltatás között: ezek mind felhasználói módban futnak.
Minden ilyen taszk beágyazódik a kernel által létrehozott hardver memória mezőjébe.
Ezen kívül más részeket csak kernel mechanizmusok révén befolyásolhat, tipikusan üzenetek küldésével. Ezek az ún**. message-passing mechanizmusok** (Inter Process Communication, IPC).

Részletesebb leírások a mikrokernelekről: pl.
https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/

Szenzorhálózatok

A szenzorhálózatokról részletes fólia-sorozat található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze.

A szenzorhálózati csomópontok jellegzetes megjelenési formája a **Berkeley Mica2 mote**, amely a fényképen látható. Mérete a nyomtatott áramköri lap alatt elhelyezett **2*AA elem** alapján becsülhető.
Felépítése és hardver jellemzői a **Szenzorhálózatok I.** című dokumentumban leírtak alapján ismerhető meg.
Ugyanitt olvasható az eszköz szóba jövő alkalmazásainak listája.
Az alkalmazások jellemzője a **térbeli kiterjedés**, és a szükséges **csomópontok nagy száma**.
A működés során lényeges az **energiatakarékosság**.

A TinyOS operációs rendszer

A TinyOS operációs rendszerről részletes fólia-sorozat található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze.

Miért van rá szükség?

A tradicionális operációs rendszerekkel nehézségek vannak szenzorhálózatok esetében, mert a **többszálas architektúra nemigen használható** kellő hatékonysággal, nagy a **memóriaigény**, az **energiafelhasználás** minimalizálását nem támogatják.

A vezeték nélküli szenzorhálózatok esetében lényeges:
(1) a konkurens végrehajtás, (2) az energiafelhasználás hatékonysága,
(3) kis memóriaigény (small memory footprint), és (4) a sokrétű felhasználás támogatottsága.

Főbb jellegzetességei:

A TinyOS nyílt hozzáférésű operációs rendszer, amely kifejezetten vezeték nélküli szenzorhálózati alkalmazásokhoz készült. Komponens alapú, NesC (Networked embedded system C) nyelven íródott a University of California, Berkeley és az Intel Research együttműködésében.

A komponens alapú architektúra lehetővé teszi a gyakori változtatásokat, és eközben a kódméret minimális szinten tartható. A végrehajtás eseményvezérelt, és ebből adódóan nagymértékben konkurens. Energia hatékony, mert a processzor - amint lehetséges – *sleep* állapotba kerül.

Kicsi a “lábnyoma”, mert FIFO alapú, nem megszakítható ütemezést alkalmaz.

Statikus memória allokációt használ, a memória követelmények fordítási időben dőlnek el.

A lokális változók mentése a stack-re történik.

Energia hatékony, kétszintű ütemezést használ:

(1) Hosszan futó taszkok és események okozta interruptok,

(2) Sleep üzemmód hacsak nincs taszk a sorbaállási sorban, ébresztés eseményre.

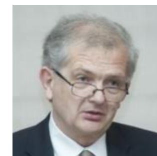
A taszkok idő-flexibilis háttér jobok, egymáshoz viszonyítva atomikusak, azaz egymás nem szakítják meg, futásukat csak interruptként megjelenő események szakíthatják meg.

Az események időkritikus, rövidebb idejű programrészek, LIFO (Last-in First-Out) logika szerint kerülnek feldolgozásra, kezdeményezhetik taszkok késleltetett futását.

A programok komponensekből épülnek fel, minden komponens specifikál egy interfészt, és ezek segítségével kerül sor a “huzalozásra”, aminek eredménye a konfiguráció.

A komponenseknek kétirányú interfészeket használnak (use), ill. biztosítanak (provide).

A komponensek parancsokat (command) hívnak és implementálnak, és eseményeket (events) jeleznek és kezelnek.

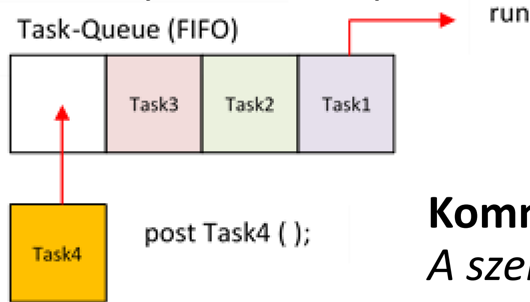


A komponensek a használt (used) interfészeken érkező eseményeket kezelik, és a parancsokat implementáló interfészeket biztosítanak (provide).

A komponensek hierarchiája:

A parancsok “lefelé” haladnak, nem blokkoló kérések, a vezérlés a hívóhoz kerül vissza.

Az események “felfelé” haladnak, taskot helyeznek el a várólistán (function queue scheduling), alacsonyabb szintű parancsot hívnak. A vezérlés a jelzést adóhoz kerül vissza.



Ciklusok elkerülésére azáltal kerül sor, hogy az események hívhatnak parancsokat, de a parancsok nem tudnak eseményt kezdeményezni.

Kommunikáció szenzorhálózatokban

A szenzorhálózatokon belüli kommunikációról részletes fólia-sorozat található a tantárgy tanszéki honlapján. Az alábbiak csak néhány kiemelt jellemzőt foglalnak össze.

Szabványos megoldások:

tipikusan az ISM (Industrial, Scientific, Medical) 2.4 GHz-es sávban, szórt spektrummal:

ZigBee/IEEE 802.15.4, IEEE 802.11b (Wi-Fi) WLAN (Wireless Local Area Network),

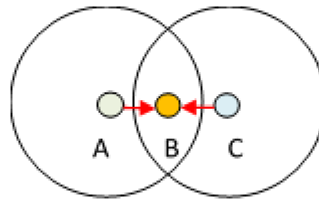
Bluetooth WPAN (Wireless Personal Area Network).

Szenzorhálózatokban tipikus a dinamikus (igény szerinti) csatorna-hozzáférési jog kiosztás, ezen belül is a CSMA: Carrier Sense Multiple Access.

Az ütközés elkerülés módja: adás előtt belehallgat a csatornába, ha nem érzékel adást, akkor adni kezd, ha adást érzékel, akkor vár.

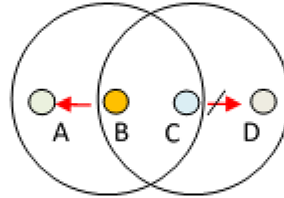
CSMA problémák:

Rejtett terminál problémája:



- A ad B-nek
- C nem hallja A-t!
- C is ad B-nek
- B egyik adást sem tudja venni

Látható terminál problémája:

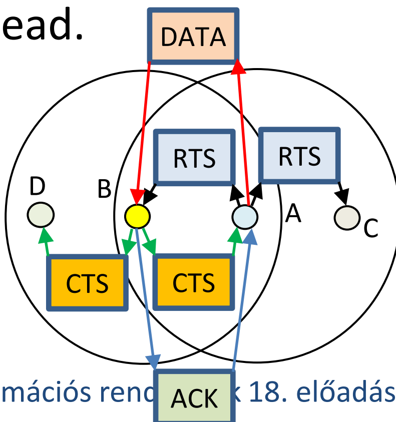


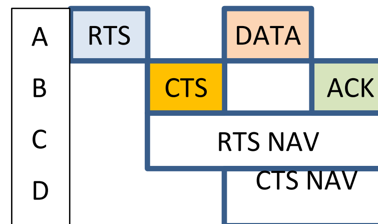
- B ad A-nak
- C is szeretne adni D-nek!
- C hallja B-t
- C nem ad, bár nem okozna ütközést

CSMA módosítások:

CSMA foglaltság jelzéssel: két csatornát használunk, az egyiket az adat továbbítására, a másikat a foglaltság jelzésére. A vevő a foglaltság csatornán folyamatosan jelez.
Adás előtt az adó ellenőrzi mind az adatcsatornát, mind a foglaltság-csatornát.
A csomópontnak egyszerre kell adni és venni, ami költséges.
Az egyidejű két csatorna nagyobb sávszélességet köt le.

Request To Send/Clear To Send (RTS/CTS): Két fázisban működik:
(1) Handshake, (2) adattovábbítás. Az alapgondolat: az ütközés a vevőnél történik.
Kizárja a rejtett terminál problémát. Hosszabb üzenetek esetén előnyös, egyébként nagy az overhead.





Az „A" adó RTS üzenetet küld ("B"-nek)
A „B" vevő CTS üzenettel válaszol
Az adó a CTS vétele után továbbítja az adatcsomagot
A többi csomópont RTS, CTS vétele után nem adhat!
(NAV = Network Allocation Vector)