Beágyazott információs rendszerek

Szepessy Zsolt (zszepes@mit.bme.hu)

Beágyazott rendszerek alapkomponensei III. : RTOS

Beágyazott operációs rendszerek

Az előző fejezet utolsó architektúrája feltételezett egy olyan software komponenst, amely az önállóan megírt taskok működését, a processzor kihasználását vezérli, a taskok közötti kommunikációt támogatja. Ez a komponens az operációs rendszer, amely bizonyos hasonlóságot mutat az asztali számítógépek operációs rendszereihez, azonban jelentősen el is tér azoktól.

Alapfogalmak

A bemutatás előtt a terminológia néhány gyakran használt kifejezése:

Task: a rendszert önálló komponensek összehangolt működésével valósítjuk meg. A teljes feladat párhuzamosan folyó részfeladatokra bontható, amelyek önállóknak tekinthetők. Ezen részfeladatok a taskok. Például egy összetett beágyazott rendszerben taskokként jelenhetnek meg az alábbiak:

- Mérőeszközük kezelését végző task
- Felhasználóval kapcsolatot tartó task
- RS485 vonali kommunikációt vezérlő task

Job: a taskok felbonthatók kis rendszeresen végzett feladatokra, ezek a job-ok. Az előző task példák esetén az alábbi job-ok elképzelhetők:

- Mérőeszközök kezelését végző task: A mérőáramkörök inicializálása, az AD konverter vezérlése, a mért adatok kiolvasása, átlagolás, erősítés beállítás stb.
- Felhasználóval kapcsolatot tartó task: billentyűzetkezelés, megjelenítés, eredmények statisztikus ábrázolása stb.
- RS485 vonali kommunikációt vezérlő task: beérkező adat kiemelése a vételi pufferből, az adatok ellenőrzése, adásnál az adási puffer olvasása (egyéb taskok töltik) és az adatok kódolása majd az adó-vevő hardware felé továbbítása stb.

Process (folyamat): önálló ütemezési entitás, amelyet az operációs rendszer önálló programként kezel. Rendszerint saját memóriaterülete van, amelyet más folyamatok nem érhetnek el. Ütemezéskor a kontextust az operációs rendszer elmenti illetve visszaállítja. Taskokat folyamatokkal implementálhatunk.

Thread (szál): olyan ütemezési entitás, amelynek nincs saját memóriaterülete. Azonos szülő folyamathoz tartozó szálak azonos memóriaterületen dolgoznak. Ennek megfelelően a kontextus-váltás gyors.

A task és job kifejezés a valós-idejű rendszerek modelljeinek tárgyalásakor, a *rendszertervezés* alatt, modellelemként jelenik meg. A process illetve thread az implementációs szakasznál a modellelemek megvalósításának eszközei.

A *beágyazott operációs rendszer* (embedded operating system) olyan software komponens, amelyet beágyazott számítógép-architektúrán futtatni lehet. Ez elsősorban a **hardware erőforrások** megfelelőségét jelenti (memóriaméret, MMU, tárkapacitás, processzorarchitektúrák). A beágyazott operációs rendszerek egy csoportja **kemény valós-idejű (hard real-time)** működést garantál (RTOS). Ez azt jelenti, hogy hardware események hatására az operációs rendszer garantált időn belül lehetővé teszi a reakciót. Ez többnyire az ütemező determinisztikus kiürítéses, prioritásos működését jelenti.

Kernel: a kernel az operációs rendszer alapvető eleme, amely a taskok kezelését, az ütemezést, és a taskok közötti kommunikációt biztosítja. Nem tartozik a kernelhez a szolgáltatások további csoportja: pl. kommunikációs csomagok, különböző hardware perifériák kezelésének rutinjai, shell stb.

A kernel implementációja hardware független (gyakran forráskódban is hozzáférhető) és hardware függő rétegekből épül fel. A hardware-függő réteg új processzorra történő adaptálását az operációs rendszer *"port"*-jának nevezik.

A beágyazott operációs rendszerek feladatai

1. Párhuzamos programozási környezet biztosítása, ami a taskok létrehozását és kezelését jelenti
2. Ütemezés: a taskok futtatása
3. Task-ok közötti kommunikáció, szinkronizálás biztosítása
4. Megszakítások kezelése
5. Időzítés
6. Memóriakezelés
7. Perifériák kezelése, rendszerprogramok (API)
8. Kommunikációs csatornák kezelése

A beágyazott operációs rendszerek és a jól ismert általános célú operációs rendszerek (desktop OS) eltérései:

Rendszerindulás:
Általános célú operációs rendszerek esetén a számítógép indulása (boot) után az operációs rendszert futtatja először a processzor, amely számos inicializáló művelet elvégzését követően biztosítja alkalmazások futtatását. Az alkalmazások relatív címeket tartalmaznak a fizikai címeket az operációs rendszer *loader* programja határozza meg az alkalmazott memóriakonfigurációval.
A beágyazott operációs rendszerek alkalmazása esetén indulás után először az alkalmazás futtatása történik. Az alkalmazás indítja el az operációs rendszert.

Programkomponensek szervezése
Az általános operációs rendszerek az alkalmazásoktól függetlenül készülnek. A lefordított operációs rendszer bináris kódot futtatja a processzor.
Beágyazott operációs rendszerek tárgykódját össze kell szerkeszteni az alkalmazással. Az operációs rendszer gyakran forráskódban adott modulok integrálásával jön létre.

Védelem
Az általános célú operációs rendszerek alapvető feladata különböző felhasználók (programok) munkájának védelme. A beágyazott operációs rendszer az alkalmazással együtt fut egy berendezésben és az alkalmazás állandó. Ennek megfelelően nem lát el hasonlóan erős ellenőrzési, védelmi funkciókat.

Skálázhatóság
Az általános célú operációs rendszerek megfelelően nagy erőforráskészlettel felvértezett számítógépeken futnak, és a szolgáltatások széles skáláját nyújtják. A beágyazott operációs rendszereknek viszonylag korlátozott erőforráskészlete van (8, 16 bites mikroprocesszorok is előfordulnak), és az alkalmazás által megkövetelt szolgáltatáscsomag a fejlesztéskor ismert. Ezért ezek az operációs rendszerek modulárisan skálázhatóak az alapvető kernelfunkcióktól a nagybonyolultságú operációs rendszerekig. A funkcionalitás fordítási időben konfigurálható.

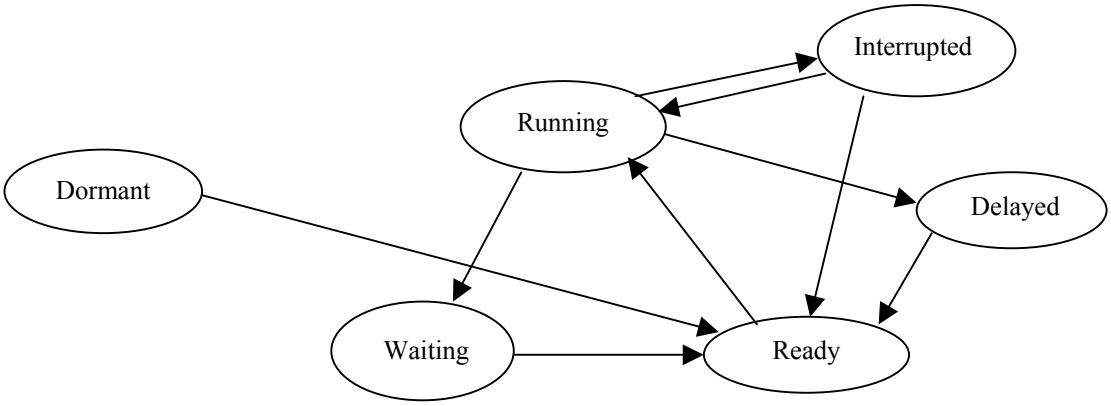
Taskok állapota

A task egy végtelen ciklus. Ahhoz, hogy a kernel más taskot is futtasson, valamilyen eseményre kell várni, kernel szinkronizálás segítségével. Ez az esemény lehet egy adott idő eltelte, vagy más tasktól vagy megszakítástól várt üzenet.

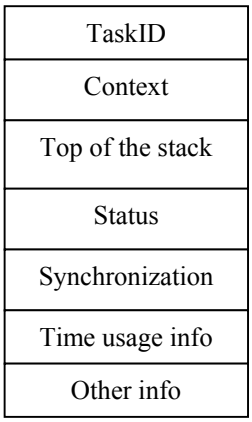
```
void Task()
        {
        while(1)
                {
                /*alkalmazás-specifikus feladat*/
                /*eseményre várakozás*/
                /*alkalmazás-specifikus feladat*/
                }
        }
```

A task az alábbi állapotokban lehet:

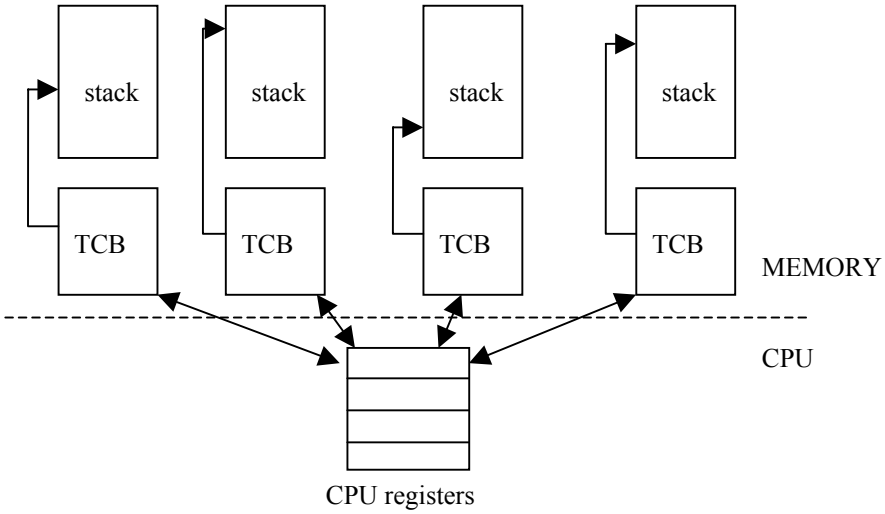
Dormant:	passzív, inicializálás előtt, vagy felfüggesztve
Ready:	Futásra kész, a prioritása alacsonyabb az aktuálisan futónál
Running:	Fut
Delayed:	Egy időintervallumra várva, rendszerint szinkron időzítő szolgáltatás hívása után
Waiting:	Adott eseményre várakozik
Interrupted:	Megszakítva (a megszakítási rutin alatt)



Task létrehozásakor meg kell adni a task működését leíró függvényt (referencia), a prioritást, és taskhoz rendelt stack terület tetejét.
A kernel a taskhoz tartozó információkat egy leíróstruktúrában tárolja (Task Control Block- TCB)

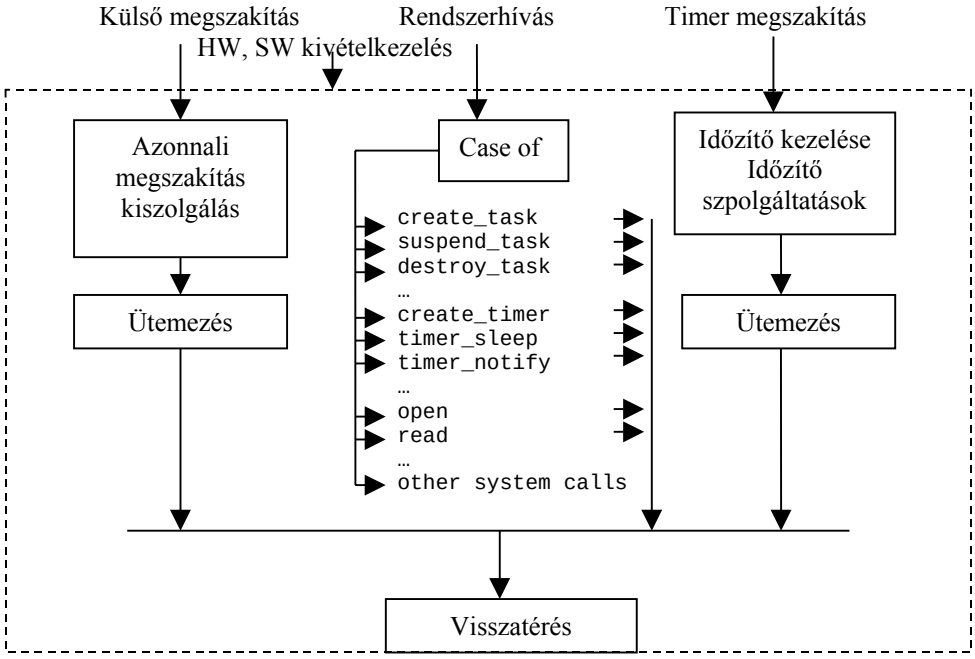


A TCB általános szerkezete



A futásra kész taskokhoz tartozó TCB táblák egy un. készenléti listába (Ready List) listába vannak fűzve. Az ütemezés feladata ezen lista tetejéről az adott task aktivizálása. Az éppen kiürített task bekerül a listába. (bonyolult felépítésű listákkal gyorsítják)

A kernel működése



Rendszerhívások (API):

A taskok a kernel adatait csak rendszerhívásokon keresztül érhetik el. Az adatok védelme gyakran memória management unit segítségével és különleges processzorállapottal biztosított. Az API hívásokra a processzor átkerül kernel módba, amellyel ezekhez a területekhez hozzáférhet. A hívás végrehajtása során:

1. a task kontextusának mentése
2. átkapcsolás kernel módba
3. hívás végrehajtása
4. visszatérés (user mode és kontextus visszaállítás) történik.

Szinkron rendszerhívás: a hívó task blokkolt, amíg a kernel el nem végzi a hívott feladatot.
Aszinkron: a hívó task folytatja a munkáját.,

Időzítő kezelés és ütemezés

A legtöbb rendszerben az ütemező periodikusan működik. Az ütemező megvizsgálja, hogy a készenléti lista (Ready List) tartalmaz-e a jelenleg futó task prioritásánál magasabb prioritású taskot, és azt futtatja. A működtetéséhez időzítő áramkört (HW timer) használnak, amely megszakításokkal szinkronizálja az ütemezőt. Az ütemezés periódusideje a *tick size.* Nagyságrendje 10 ms.
Az időzítő megszakítás hatására az alábbi kernel működés indul:

1. Időzítő események feldolgozása: Az időzítőhöz rendelnek egy *timer queue*-t, amelybe azok az időtartamok kerülnek, amelyeket a taskok kértek. A kernel megvizsgálja, hogy az előző esemény óta van-e olyan időkérés, amely éppen aktuális. Ekkor az eseményre várakozó taskoknak jelzést küld (az esemény queue-kat feltölti).

2. Módosítja a jelenleg futó task időszámlálóját. Az operációs rendszerek biztosítják az azonos prioritású taskok ciklikus (round-robin) jellegű ütemezését azonos időszelettel. Ez az ütemezés ezen a számlálón alapul.

3. Aktualizálja a készenléti listát (Ready List), azaz végrehajtja az ütemezést

Külső megszakítások

A megszakítások kiszolgálása két lépésben történik:

1. Azonnali megszakítás kiszolgálás

Az operációs rendszer prioritási rendszere:
Legmagasabb prioritású:
- Rendszerleállítás
- Tápfeszültség kimaradás
- Időzítő megszakítás
- Legmagasabb HW megszakítás
- …
- Legalacsonyabb HW megszakítás
- Ütemező prioritása
- Legmagasabb prioritású task
- …
- Legalacsonyabb prioritású task

A megszakítás észlelésekor a processzor menti a programszámlálót és a processzor állapotát a verembe (stack), majd elugrik a kernel megszakítás-kezelő rutinjára. A kernel működése közben a további megszakítás tiltott. A kernel elmenti az aktuális task adatait, engedélyezi a megszakításokat és meghívja az adott megszakítás-kezelő függvényt.
A megszakítás kiszolgálás teljes késleltetése (interrupt latency) a következő időtartamokból áll össze:
1. Processzor utolsó utasításának befejezése
2. A megszakítások tiltásának ideje
3. A magasabb prioritású megszakítások kiszolgálásának ideje
4. A megszakított task kontextusának mentési ideje
5. A megfelelő megszakítás-kezelő rutin indítása

2. Ütemezett megszakítás kiszolgálás
A megszakítás-kezelés gyors első lépéseként az előző pontban ismertetett kiszolgálás történik, második lépésként pedig egy olyan kiszolgáló rutin, amely *kiüríthető* és a normál ütemezési rendszerben kezelendő. Prioritása többnyire azon task prioritásával egyezik meg, amely megnyitotta a hardver eszközt. Az ütemezett megszakítás-kezelő rutinokat kernel taskok hajtják végre, hiszen azok kernel adatokat kell elérjenek.

Más egyszerűbb operációs rendszerek lehetővé teszik saját megszakítás-kezelő rutinok alkalmazását. Előírják viszont, hogy ilyen esetben a kernelt informálni kell arról, hogy éppen megszakítás végrehajtás van illetve, amikor a kezelőrutin végén is jelzést kell adni. A kernel az megszakítás-kezelő rutinba belépéskor egy *interrupt nesting* paramétert növel, kilépéskor csökkent.

Ütemezés

Minden elérhető beágyazott operációs rendszer biztosítja az **állandó prioritású ütemezést**. Legtöbb operációs rendszer 256 prioritási szint beállítását kínálja. (általános OS többnyire 16 szintet) A task létrehozásakor adott prioritás a hozzárendelt prioritás (assigned priority). Egyes rendszerekben működés közben egy task a hozzárendelt prioritásánál nagyobb prioritást is örökölhet (priority ceiling protocol, priority inheritance). Az aktuális prioritás (current priority) tehát eltérhet a hozzárendelt prioritástól. A kernel minden prioritási szinthez létrehoz egy készenléti listát. A legmagasabb prioritású készenléti task megkeresése ebben az esetben a legmagasabb nem-üres lista megkeresését jelenti.
Az **azonos prioritású** taskok ütemezéséhez **round-robin** vagy **FIFO** ütemezéseket kínálnak a kapható rendszerek.
A legtöbb operációs rendszer az állandó prioritási séma mellett lehetővé teszi a **prioritás dinamikus változtatását**. Ez egy adott rendszerhívás segítségével valósulhat meg.

A fejlett RTOS biztosítja a taskok számára a kiürítés tiltását (*preemption lock* - `taskLock()`, `taskUnlock()` (`VxWorks`)).

Egyes operációs rendszerek (pl. QNX) lehetővé teszik a taskok különböző processzorokra ütemezését. Ilyenkor az összes készenléti task egy listába kerül és a legmagasabb prioritásút az ütemező bármely lehetséges processzorra irányíthatja. A lehetséges azt jelenti, hogy a taskok TCB leíró táblájában van egy mask szó (*processor affinity*), amely jelzi, hogy mely processzorokra ütemezhető a task.

Időzítési szolgáltatások

Az operációs rendszerek mindegyike legalább egy óraegységet működtet (az ütemezőt működtető időzítőt). Az óraegységhez tartozik egy

- Számláló
- Időzítő queue
- Megszakítás-kezelő rutin

A rendszeridőt minden időpillanatban a számláló állása adja. A queue-ban a taskok által kért időzítési kérések sorakoznak.
A legtöbb operációs rendszerben minden taskhoz tartozhat egy saját *software timer*. Ez egy adatszerkezet, amit a kernel hoz létre a `create_timer()` jellegű rendszerhívásra. A timert egy adott fizikai órához kell rendelni, ha a rendszerben különböző órák vannak. A struktúra tartalmaz egy kezelő függvény-referenciát azzal a céllal, hogy a timer eseménynél a kernel ezt automatikusan meghívja.
A timerek lehetnek egyszeri lefutásúak, periodikusak, abszolút időjelöléssel illetve relatív időzítéssel.

Aszinkron timer szolgáltatások:
Az aszinkron timerek beállítás után elindulnak, közben a hívó task folytatja a munkáját. A timer szolgáltatások között a megbízhatóság egyik fontos eszköze a watchdog timer. Ez egy aszinkron timer, ami létrehozása után bármikor elindítható egy adott függvényreferenciával és egy paraméterrel. A timer lejárta előtt egy másik rendszerhívással kikapcsolható. Az eszköz hatékonyan támogatja elakadás, lefagyás, határidő-túllépés megakadályozását.
A timer esemény egyes esetekben paraméterként megadott függvény hívásával, más esetekben üzenetekkel történik. (POSIX signal-t küld)

Szinkron timer szolgáltatások:
A szinkron timer szolgáltatások hatására a hívó task Delayed állapotba kerül.

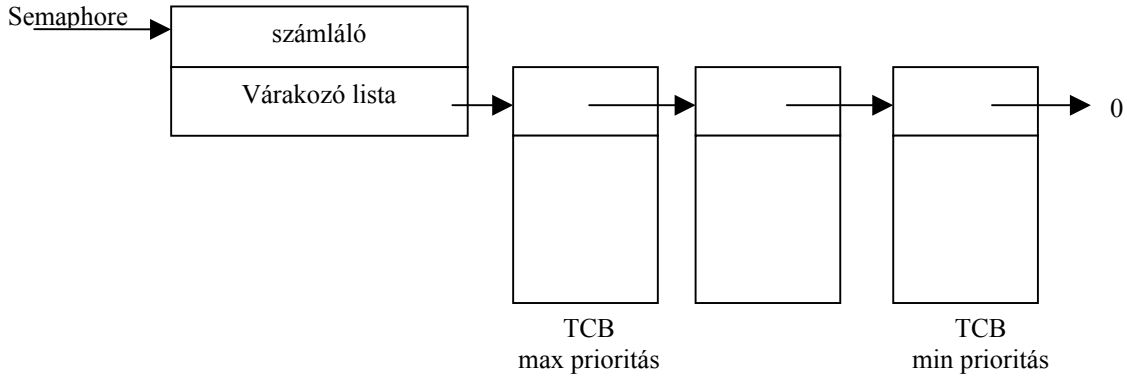
Taskok közötti kommunikáció és szinkronizálás

Szemafor

A szemafor minden operációs rendszer által biztosított elemi szolgáltatás.

Célja
- megosztott erőforrásokhoz való hozzáférés vezérlése,
- események jelzése,
- taskok tevékenységének szinkronizálása.

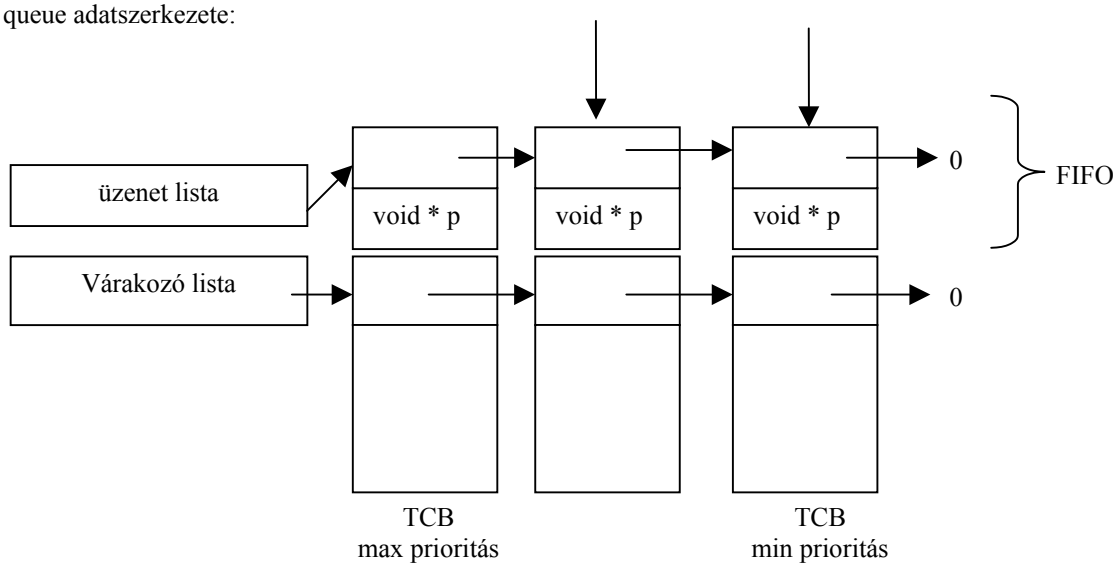
A szemafor olyan kulcs, amire a tasknak szüksége van a továbblépésre. Amennyiben a kulcsot éppen használja valaki, a task várakozásra kényszerül. A szemafor leggyakoribb felhasználása kölcsönös kizárás biztosítására történik. Megvalósítását tekintve egy lehetséges forma a következő: a szemafor egy 16 bites számot és egy várakozási listát tartalmaz. A szemafor használata előtt a létrehozó rendszerhívással inicializálni kell az értékét. Amennyiben valamely task használni kívánja a szemaforral védett erőforrást (pl. printer), akkor egy PEND utasítást hajt végre. Amennyiben a szemafor értéke nagyobb, mint 0, akkor az azt jelenti, hogy az erőforrás szabad, a task hozzáférhet. A kernel a szemaforhoz tartozó adatszerkezetben az értékét csökkenti 1-el. (egyszerű esetben ekkor a szemafor értéke 0) Ha egy másik task is el akarja érni az adott erőforrást, akkor a PEND utasításra blokkolódik, hiszen a szemafor értéke ekkor már 0. A task kikerül a készenléti listából és bekerül a szemaforra várakozók listájába. Az erőforrás felszabadítása után a birtokló task egy POST utasítást ad ki, amellyel a szemafor értékét növeli 1 el. Ennek hatására a várakozó listában lévő taskok közül a legmagasabb prioritású átkerül a készenléti listába. A szemafor változót a kernel kezeli, így az speciális állapotban látható (pl. ARM processzornál Supervisor mód).



QUEUE

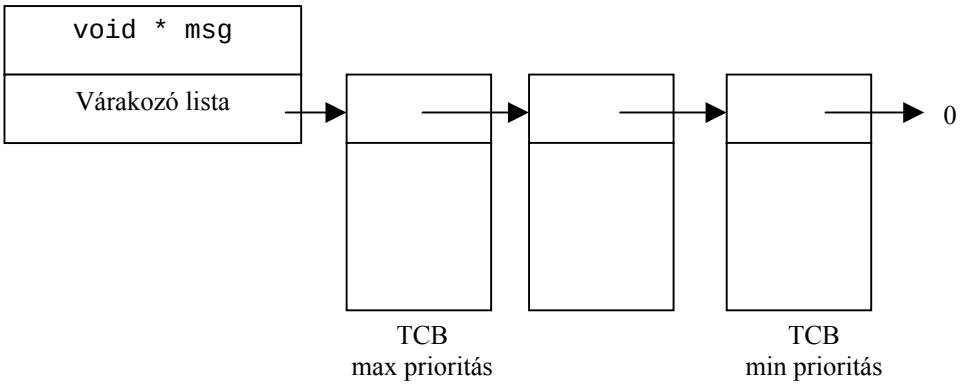
A queue egy FIFO (first in first out) szervezésű lista, amelybe egy vagy több task üzeneteket írhat más taskok számára. A queue-k könnyen implementálhatók hálózati rendszerekre is. A task számára ugyanazt a felületet jelenti, mintha a címzett ugyanazon a processzoron futó másik task lenne. A leggyorsabb működést biztosító valós-idejű kernelek csak FIFO jellegű queue-t kínálnak. Összetettebb rendszereknél különböző prioritásos kiemelés is megjelenik. Itt már nehezen eldönthető, hogy az adott kommunikációs csatorna valójában egy queue vagy mailbox (jellegzetesen azonos generikus típusú entitások pl. OS_EVENT). A prioritásos queue-t biztosító operációs rendszerek között van, amely két prioritási szintet biztosít: normális, sürgős. A POSIX kompatibilis rendszerek 32 prioritási szint megadását teszik lehetővé. Az üzenetet ki kell olvasni a megfelelő rendszerhívással.
Alapvető műveletek: POST: queue-ba írás, PEND: queue-ból olvasás. Ez utóbbi művelet blokkolja a hívó taskot, ha a queue üres. A queue-ba írás nem blokkol. Ameddig van hely a queue-ban, a küldő task tovább futhat. A megtelt queue esetében blokkolhat (ez operációs rendszertől függ).
A kernel jelzi a várakozó taskoknak, ha üzenet érkezett a queue-ba. A blokkolt taskok így a készenléti listába kerülnek.
A queue implementációját tekintve egyszerűen egy referencialistát (void * msg) és a queue-ra várakozó taskok listáját jelenti. A referencialista az üzeneteket tartalmazza. A taskok a queue-ba helyezik az általuk küldeni kívánt üzenetre mutató referenciát. A mutatók kiemelése és interpretációja a vevő task feladata. Létezik nem blokkoló hatású függvény, amely megvizsgálja, hogy található-e üzenet a queue-ban.

A queue adatszerkezete:



MAILBOX

A mailbox olyan üzenetküldési mechanizmus, ahol az üzenet egy void pointer lehet, ami az aktuális processzorra jellemző méretű adat. Alapvető különbség az előzőhöz képest, hogy hagyományosan a mailbox-ban egy üzenet van.
Alapvető műveletek: POST- mailbox-ba írás és PEND mailbox-ból olvasás. Az olvasás blokkol ha üres a mailbox. A megtelt mailbox-ba történő írás vagy blokkol, vagy felülírja a jelenlegi adatot.
Megvalósítását tekintve a szemaforhoz hasonlít: egy void típusú referencia és egy task várakozó lista. A mailbox-ból olvasó blokkolt taskok a listába kerülnek prioritási sorrendben.
Ha üzenet érkezik, a legnagyobb prioritású task átkerül a készenléti sorba. Egyes operációs rendszereknél a mailbox-ba több üzenet is írható és az üzenetek prioritással rendelkeznek.
Létezik nem blokkoló hatású függvény, amely megvizsgálja, hogy található-e üzenet a mailbox-ban.



PIPE

Byte szervezésű csatorna a queue-hoz hasonlóan. FIFO módon kezelhető fread(), fwrite() standard függvényekkel. Egyes operációs rendszerek megengedik különböző hosszúságú üzenetek írását (a queue és a mailbox egy void referenciát engedett küldeni). Az üzenetek interpretációja az alkalmazás

feladata. Ha A beleír a pipe-ba 11 karaktert, B vevő pedig kiemel 13 karaktert, akkor azzal egy másik üzenetet is megbont.

EVENTS

Az események analóg módon működnek a megszakításokkal. Tulajdonképpen egy olyan flag, amelyet egy task bekapcsolhat vagy kikapcsolhat, és amelyre egy másik task várakozhat. Egyes rendszerekben a terminológia jelzést (*signal*) használ ugyanerre a mechanizmusra.
Egy adott eseményre egyszerre több task is várakozhat. Az eseményekből az operációs rendszerek rendszerint eseménycsoportokat alkotnak (*event group*), amelyeknek maskokkal meghatározott részhalmazára egy task blokkol. Az események kikapcsolására a különböző rendszerek eltérő megoldást kínálnak: egyes esetekben automatikus más esetekben a task által elvégzendő.
Az eseménykezelés történhet a UNIX rendszereknek megfelelően: a taskoknak az adott eseményhez rendelt kezelő függvényeik vannak, amelyek az esemény bekövetkezésekor a kernel által aktiválódnak. Más esetekben egy WAIT jellegű függvényhívással a hívó task blokkol az esemény bekövetkeztéig.
POSIX kiterjesztések:

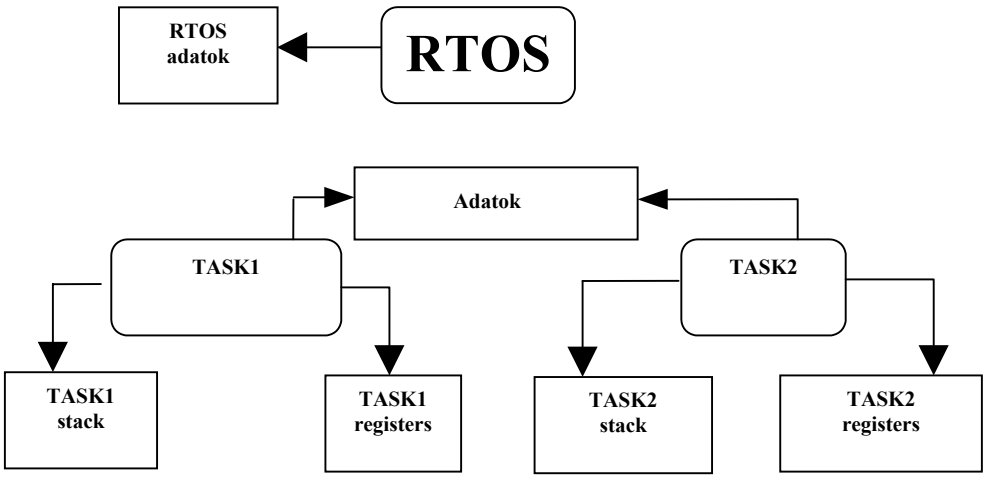
- Legalább nyolc felhasználó által használható jelzés.
- A jelzések queue-ban gyűjthetők.
- Az üzeneteknek értéke is lehet.
- Az eseményeknek prioritása van.
- Az eseménykezelő függvény a blokkolt task visszaállítása előtt fut.

Memória kezelés

Az operációs rendszereket három csoportba lehet sorolni:

1. Nincs virtuális memóriakezelés
   Ez a legegyszerűbb eset, akkor használatos, ha a teljesítmény nagyon kritikus. Virtuális memóriakezelés nélkül a task-ok egy összefüggő fizikai címtartományban kapják meg a szükséges memóriát (változók, stack). A kért blokkok fizikailag folytonosak. Ezért ezekben a rendszerekben gyakran töredezett a memória, így gyakran megnő a kihasználatlan tárméret. (feltéve, hogy a memóriaigény működés közben változik). Fontos jellemzője ennek a csoportnak, hogy a taskok azonos címtérben vannak, gyakran a kernellel is. A memóriakezelés alapeleme a *pool.* A pool azonos hosszúságú memóriablokkokat (*buffer*) tartalmaz. A taskok kérhetnek a pool-ból újabb blokkokat. Amennyiben az adott pool nem tartalmaz újabb szabad memóriaterületet a hívó task blokkolt állapotba kerül.
   Az operációs rendszer "nem tudja", hogy az adott memóriaterület fizikailag hol van. Ezért az operációs rendszer indításakor (ez az alkalmazás dolga) meg kell adni a megfelelő memóriaterületeket.

   Jellegzetes memóriakonfiguráció:



2. Lapozás biztosítása

A lapozással (*paging*) megoldható, hogy a kritikus valós-idejű és az egyéb nagy memória-igényű háttérfolyamatok különböző tárterületeken fussanak. Működés közben a tárterületek nem befolyásolhatják egymást.

3. Virtuális memóriakezelés memóriavédelemmel

Egyes operációs rendszerek bonyolult memóriavédelmet kínálnak. A task-okhoz rendelt területeket HW MMU egység segítségével védik egyéb taskoktól. Hasonlóan védett a kernel adatterülete, az interrupt vektortábla stb.

Biztonságkritikus operációs rendszerek

Az előző fejezetben ismertetett általános jellemzőkön túlmenően (illetve azokkal ellentétben) a biztonságkritikus rendszerekben alkalmazott operációs rendszerekkel szemben az alábbi követelmények támaszthatók.

1. Memóriavédelem

Elengedhetetlen az MMU használata. Nem futhatnak a taskok és a kernel azonos címtartományban. A virtuális memóriakezelést támogató operációs rendszerek sokkal nagyobb biztonságot jelentenek. A memóriakonfigurációt úgy kell beállítani, hogy a stack túlcsordulás HW jelzést okozzon, aminek hatására a kernel hibakövető taskot futtathat.

2. Hibatűrés

A kernel ellenőrzi a taskok működését. Ha egy task hibásan dolgozik, a kernel hibajavító, felügyelő taskot futtat. A felügyelő task elkülönített címtartományban működik.
A kernel az alábbi hibakezelési szolgáltatásokat nyújtja:

- A hibás task leállítása
- Az alkalmazás újraindítása
- Események gyűjtése, naplózása
- Software watchdog
- Illegális rendszerhívásokkal szembeni önvédelem

3. Redundancia

A feladatot elvégző együttműködő komponensek redundáns megvalósításával lehetővé válik, hogy valamely komponens kiesése esetén a tartalék komponens folytatja a munkát. Ehhez egyfajta "szívverés-szerű" kommunikációs csatornát alakítanak ki komponensek és a redundáns elem között. Ha egy adott időtartam alatt nem érkezik szívverés egy komponenstől, akkor annak helyére áll be a redundáns elem.

4. Hozzáférések ellenőrzése

A kernel szigorúan ellenőrzi kritikus rendszereszközök elérését. Egyes eszközöket csak adott taskokhoz rendeli. Ezekhez az eszközökhöz más nem férhet hozzá.

5. Garantált erőforrás-biztosítás: memória

Az operációs rendszer biztosítja, hogy a dinamikus tárigények nem érinthetik a kritikus taskok memóriaterületeit illetve a kernel központi adatterületét.

6. Garantált erőforrás-biztosítás: processzoridő

A processzorciklus ablakokra van bontva, amelyekben csak adott task-csoportok ütemezhetők. Így kritikus taskokhoz rendelhető garantált processzoridő.

7. Ütemezés

Az operációs rendszer rendszerhívásainak garantált időkorlátja lehetővé teszi a determinisztikus működést. Az un. *overhead* becsülhető kell legyen. A prioritásöröklés kiszámíthatatlan futási időt eredményez ezért a HL (highest locker) szemaforok alkalmazása ajánlott. Ez olyan szemafor, amelynek hozzárendelt prioritása van: a működés során a szemafort igénylő taskok közül a legmagasabb prioritásúval megegyező prioritás. Ha egy task megkapja a szemafort, akkor a saját prioritását a szemafor prioritására emeli.

Valós-idejű operációs rendszerrel együttműködő software tervezési kérdései

A megszakítás-kezelő rutinok (ISR) írásának szabályai

1. A szabványos C-ben nem találunk nyelvi elemet a megszakítás-kezelő függvények megjelöléséhez. Így a kijelölésükhöz a következő lehetőségeink vannak:

a.) A megszakítás-kezelő rutinokat a többi függvényhez hasonlóan definiáljuk, és a program inicializáló szakaszában a fenti példához hasonlóan a processzor megszakítás-vektortábláját úgy állítjuk be, hogy a kívánt függvények címeit tartalmazza. Ebben az esetben azonban arra kell ügyelni, hogy a függvény elvégezze a megszakítás-kezelésnél előírt műveleteket, és a visszatérésnél a megfelelő helyre kerüljön a vezérlés.

b.) A processzorokhoz szállított C fordítók kiterjesztéseket tartalmaznak a megszakítás-kezelő függvények deklarációjához. Például:

```
_irq void IRQHandler (void ) {...}
void interrupt IRQHandler(void) {...}
void IRQHandler(void) interrupt 4 {...}
```

A direktíva hatására a C fordító a függvényünket kiegészíti az alábbiakkal:

- processzor regisztereinek mentése
- a függvény által használt regiszterek mentése
- kilépéskor a regiszterek visszaállítása
- ezek a függvények nem újrahívhatók

A megszakítás-kezelő rutinok írását támogató C fordítók használata esetén is nekünk kell assembly nyelven megírni a függvényeket, amennyiben:

- újrahívható függvény kell. Ennek oka, hogy a processzorok alapértelmezés szerint a megszakítás-kiszolgálás elején tiltják a további hasonló megszakításokat. Ezeket a megfelelő helyen engedélyezni kell.
- software megszakítás kiszolgálása, hiszen a software megszakítás, mint utasítás tartalmazza a megszakítási okot, amelyet az utasítás kódjából kell kifejteni. Ekkor a kiszolgáló rutinban ki kell deríteni melyik utasítás tartalmazta a megszakítást és ennek az utasításnak az elemzésével dönthető el a megszakítási ok.

2. Megszakítás-kezelés szabályai RTOS környezetben

SZABÁLY 1:

A megszakítás-kezelő rutinban hívhat olyan RTOS függvényt, amely blokkolja a hívót:

- nem kérhet szemafort (uC/OS: OSSemPend)
- nem várhat eseményre

Amennyiben ugyanis a rutin blokkoló utasítást hívna, akkor az blokkolná a megszakított taskot is, és a megszakítás-kezelő rutint nem lehetne befejezni, amely katasztrófális következményekkel járhat. (beágyazott környezetben a program "fagyása" mellett nagyon veszélyes hatásai lehetnek a nem megfelelő működésnek.)

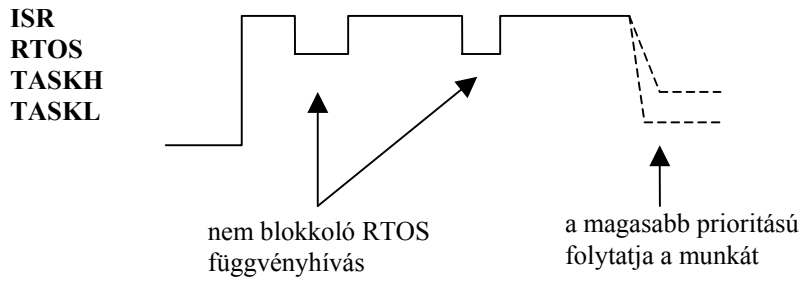
Meg kell jegyezni, hogy az operációs rendszerek többnyire kínálnak olyan függvényeket is, amelyek nem blokkolják a hívót (szemafor értékének lekérdezése, esemény queue lekérdezése ...)

SZABÁLY 2:
A megszakítás-kezelő rutin nem hívhat olyan RTOS függvényt, amely task-váltást eredményezhet, KIVÉVE akkor, ha az RTOS "tudja", hogy megszakítás-kezelő rutin fut.
Azaz:

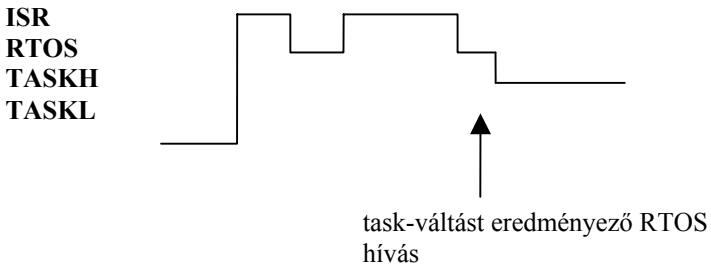
- nem írhat mailbox-ba, queue-ba
- nem engedhet el szemafort
- nem küldhet eseményt

Amennyiben ugyanis a tiltott hívások valamelyikével élne a rutin, a megszakított task-nál magasabb prioritású task kerülhet futásra kész állapotba, blokkolva ezzel a megszakítás-kezelő rutint és a megszakítás-vezérlő hardware-t.
A jelenséget szemlélteti az alábbi ábra:

HELYES MŰKÖDÉS



HELYTELEN MŰKÖDÉS



Felmerül a kérdés, hogy milyen módon tud a szabályok betartásával ezek után egy megszakítás-kezelő rutin részt venni a taskokban megfogalmazott rendszer működésében. A válasz a második szabály kivétel részében rejtőzik: biztosítani kell, hogy a valós-idejű operációs rendszer tisztában legyen azzal, hogy a task-váltást eredményező hívások egy megszakítás-kezelő rutintól jönnek. Ilyen esetben az RTOS módosítja a taskok készenléti sorait, de a task-váltást csak a megszakítás-kezelő rutin befejezése után engedélyezi. Úgy is fogalmazhatunk, hogy a megszakításkezelés alatt az ütemező tiltott állapotban van. Az RTOS informálásának két módszere ismeretes:

- A legtöbb RTOS a megszakításkezelést úgy támogatja, hogy a tervező a kódban az inicializáló szakaszban "bejegyzi" a kiszolgáló rutinokat megfelelő RTOS függvényhívásokkal. Ezek után a megszakításkezelést teljesen az operációs rendszer vezényli, vagyis a megszakítás hatására először az RTOS megfelelő rutinja fut, amely tiltja az ütemezőt, elvégzi a kötelező mentéseket stb., majd a megszakításhoz tartozó bejegyzett függvényt hívja.
- Egyszerűbb esetben az RTOS biztosít olyan függvényhívásokat, amelyek a megszakításkezelő-rutinba való belépést és kilépést jelzi (`EnterISR()`, `ExitISR()`). Ezeket a függvényhívásokat a megszakítás-kezelő rutinban alkalmazni kell a megfelelő helyeken. Ezzel a módszerrel sokkal gyorsabb és közvetlenül vezérelhető megszakítás-kezelést érhetünk el, igaz a programozói hibák valószínűsége is nagyobb.

Task-ok írásának szabályai

Az egyik alapvető kérdés a valós-idejű operációs rendszerrel együttműködő beágyazott software írásakor a kialakított taskok száma. A kérdés alapvetően modellezési probléma, de a beágyazott környezet - mint minden egyéb tervezési kérdésnél - erősen befolyásolja a döntést.
Sok task kialakításának előnyei:

- A software a környezet eseményeire mutatott válaszideje finoman hangolható. Az RTOS alkalmazása mögött az elsőszámú indok a kiürítéses működés, tehát a prioritások hozzárendelésével a válaszidő jól kezelhető.
- A taskok támogatják a software egyes elemeinek egységbezárását, így ez a moduláris felépítés hatékony eszköze.
- A különböző hardware komponensek tervezési részletei jól elfedhetők a megfelelő taskokban. A kód újrahasználhatóságát is támogatja a task szervezés.

Sok task kialakításának hátrányai:

- A több task több taskok közötti kommunikációt jelent, amely RTOS hívásokkal valósul meg. Az RTOS hívások lassítják a kódot.
- A taskok számával a szükséges szemaforok száma is nő, amely a programozási hibák valószínűségét növeli, illetve lassítja a kódot.
- A taskok működéséhez stack tartozik, amelyek a memóriaigényt növelik.
- A taskok számával a szükséges kontextus-váltások száma is nő, amely szintén a program futását lassítja.
- A valós-idejű operációs rendszerek megadják a maximális hívási gyakoriságot, amely szintén tervezési korlátként jelentkezik.

A fenti szempontok közül fontos hangsúlyozni, hogy az előnyök csak gondos tervezés esetén, a hátrányok viszont mindig jelentkeznek. Megállapítható, hogy a lehető legkevesebb task alkalmazása javasolható. Amennyiben egy rendszer kialakítása során az RTOS alkalmazása elkerülhetetlennek tűnik, célszerű az alkalmazott operációs rendszert a lehető legkevesebbszer hívni.

Az önálló task alkalmazásának indokolt esetei:

- Az egyes tevékenységek kiürítéses működésének biztosítására megkülönböztetett prioritású önálló taskokat kell létrehozni. Ez a software valós-idejű követelményeinek teljesítése érdekében szükséges.
- A megosztott hardware komponensekhez önálló taskokat kell rendelni. (nyomtató, kijelző, flash memória ...)
- Minden külső eseményhez önálló taskot kell rendelni
- A taskok felépítése egyszerű kell legyen. Összetett taskok helyett több egyszerű task alkalmazása javasolható.

Ajánlott task felépítés

```
task_A.c

/*private static data*/
void vTaskA (void)
{
/*Private data*/
/*static stack*/
while (FOREVER)
      {
      /*wait for a signal, msg_box, event, queue ...*/
      switch (/*type of signal*/)
            {
            case /*signal type 1*/
            break;
            case /*signal type 2*/
            break;
            }
      }
}
```

- Az ajánlott task felépítés legfontosabb jellemzői:
- Minden task önálló C forrásfile-ban van.
- A task csak egy helyen blokkolt.
- Ha a tasknak nincs dolga, akkor várakozik, nem használja a processzort.
- Nem használ más taskok által is látható publikus adatokat.
- A task működése jól leírható véges állapotú automatával (FSM).

A kommunikáció egységbezárása

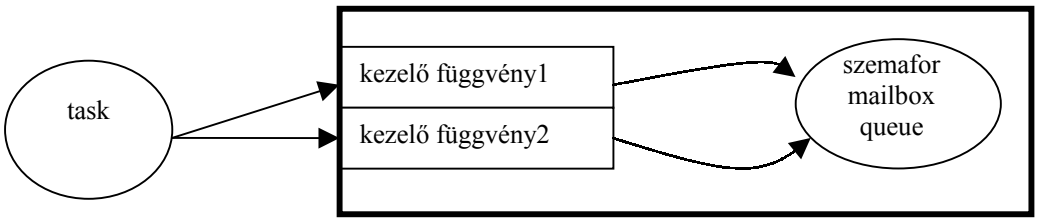
Az egyes taskok közötti kommunikációs módszerek számos programozási hibát csempészhetnek a software-be. A beágyazott valós-idejű operációs rendszerek a hatékonyság, sebesség és a kódméret miatt komoly felelősséget hárítanak a tervezőre. A leggyakoribb hibák a szemaforok esetén a következők:

- elfelejtett kérés vagy elengedés
- nem a megfelelő szemafor kezelése
- túl sokáig foglalt szemaforok
- prioritás inverzió - deadlock

A queue illetve mailbox használatának gyakori hibái:

- a task nem a megfelelő queue-ba ír
- a queue-ba írt adatok rossz értelmezése
- queue-ba helyezett referenciák (pointer) kezelésének hibái

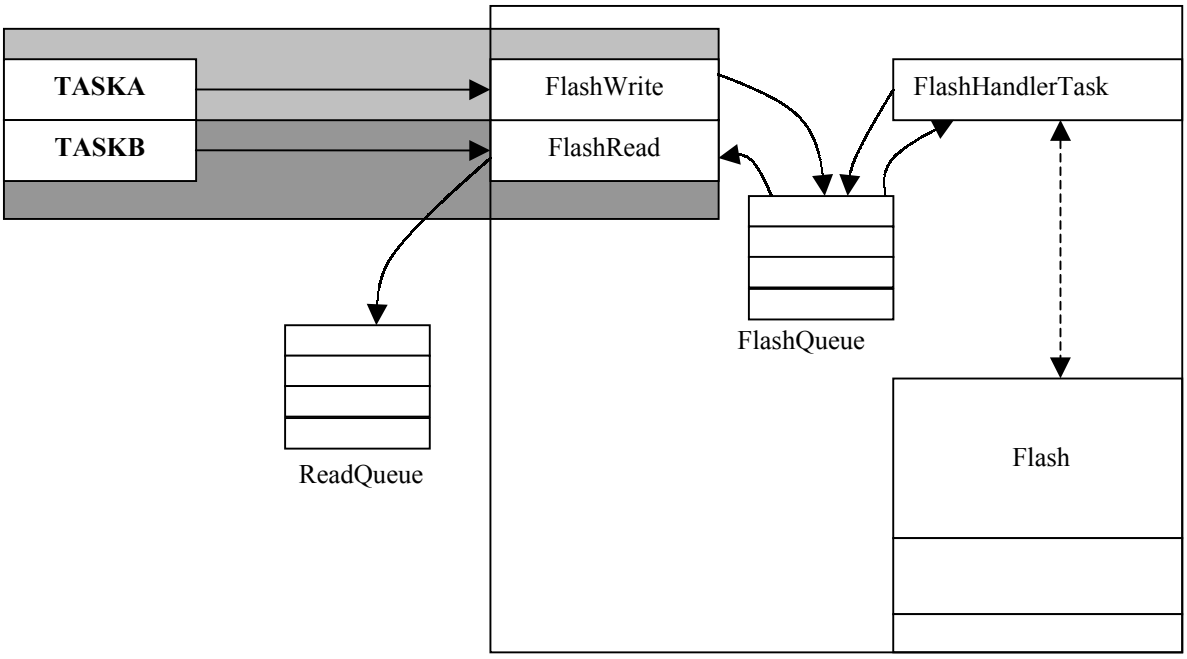
Mindezen hibák valószínűségének csökkentésére célszerű a kommunikációs csatornákat függvényhívások mögé rejteni. Ezzel a kód ellenőrzését egy helyen kell megtenni, és a javítása is könnyen elvégezhető. Az egységbezárás vázlata a következő:



Az alábbi kódrészlet a megosztott hőmérsékletadat kiolvasása esetén rejti el a hívó elől a szemafort.

```
int TempGet()
      {
      int t;
      OSSemPend(SEM_TEMP);
      t=TempHWRead();
      OSSemPost(SEM_TEMP);
      return t;
      }
```

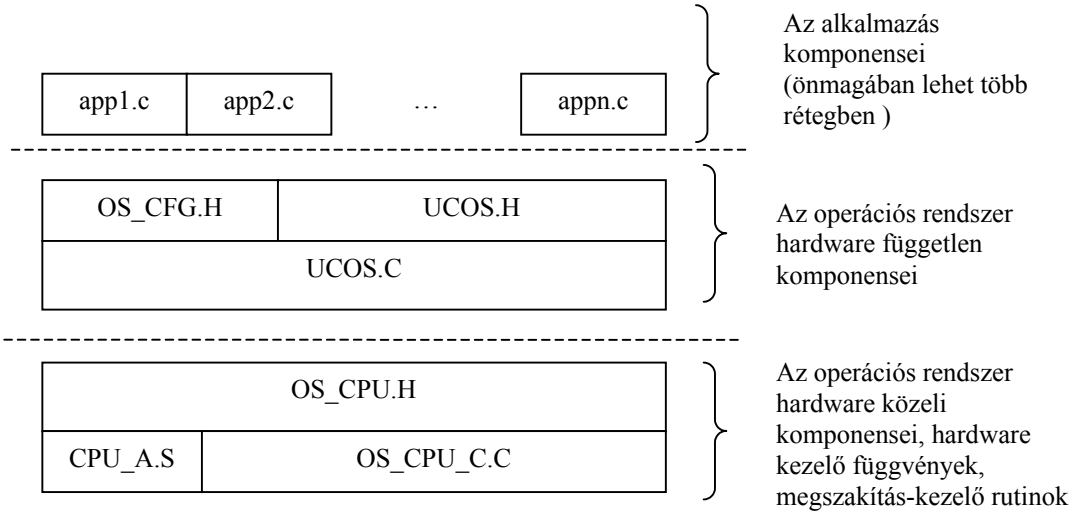
Az alábbi ábrán Flash memóriához tartozó queue egységbezárása látható:



A rendszer működése a következő: A Flash kezelése speciális parancsokkal lehetséges. Egyszerre csak egy task kéréseit lehet a Flash felé végrehajtani, ezért a Flash kezelését önálló task végzi (FlashHandlerTask). Ez elvégzi a Flash-kérések kölcsönös kizárását. Az általa nyújtott szolgáltatások a bemeneti queue-ba érkező kérésekkel érhetők el. Ebbe a queue-ba olyan struktúrákra mutató referenciákat kell helyezni, amely struktúrák tartalmazzák a kérés azonosítóját (írás. olvasás, törlés...), illetve a kérés paramétereit, adatait (címtartomány, adatok...). Olvasási kérés esetén meg kell adni annak a queue-nak a címét is ahova a kiolvasott adatokat kell helyezni. A példában TaskB fordul olvasási kéréssel a FlashHandlerTask-hoz és megadja a ReadQueue-t, mint a kiolvasott adatok helyét. Nyilvánvaló hogy az összetett parancsformátum miatt, és a kényelmesebb programozás miatt célszerű két függvényt definiálni, amelyek a szolgáltatásokat a külső taskok felé biztosítják. Ezen függvények paraméterei értelemszerűen írás esetén a cím és adat, olvasás esetén a cím és a cél-queue. A parancsstruktúra felépítése és queue-ba helyezése a függvények dolga. Az ábrán a sötét mezők azt jelzik, hogy az ábrázolt szituációban a meghívott függvények a hívó task kontextusában futnak.

Beágyazott software rétegszerkezete

A software-ek rétegszerkezete az alkalmazás szempontjából hasonlóan alakítandó ki, mint ahogy azt az általános szoftvertechnológia módszerei előírják. Figyelembe kell ugyanakkor venni a korlátozott erőforráskészletet. A rétegszerkezet szempontjából ez elsősorban a kódméretet és a stack memória méretet jelenti. Jellegzetessége ezen rendszereknek továbbá a hardware - hez közeli rétegek és a beágyazott operációs rendszerhez tartozó rétegek jelentősége. Az alábbi ábra egy konkrét rendszert mutat be, amely jellemző a beágyazott rendszerekre.



Az alkalmazás rétegében találhatóak a tervező által készített forrásállományok. A beágyazott software-ek nagy része C nyelven egy jelentős része gépi (assembly) nyelven íródik. Egy kis résznél található a C++ és újabban *néhány* alkalmazásban a Java. Az alkalmazás komponenseit a tesztelhetőség figyelembevételével kell elkészíteni (l. későbbi előadások).
Az operációs rendszer API készlete definíciós file-okban adott. A valós idejű operációs rendszerek skálázható tulajdonságait a forrásfile-ok (vagy bináris tárgykódban adott könyvtárak) statikus, de szelektív szerkesztésével érik el. Ez praktikusan azt jelenti egy forráskódban adott operációs rendszer esetében, hogy az egyes függvények (pl. semaphore kezelés, queue kezelés stb.) külön-külön forrásfile-ban és önálló definícós állománnyal szerepel. Az ábrán egy forrásfile-t (UCOS.C) és a hozzátartozó definíciós file-t (UCOS.H) ábrázoltam. Az OS_CFG.H azon konstansokat, definíciókat tartalmazza, amelyek az operációs rendszert konfigurálják (elemek maximális száma egy queue-ban, mailbox használata vagy kikapcsolása, általános konstansok: TRUE, FALSE, MAX_PRIORITY stb.). Az alsó rétegen az operációs rendszer hardware függő komponenseit találjuk. Egy rendszer esetén új hardware-re való áttéréskor ezen komponensek adaptálására van szükség. (ez legtöbbször az új hardware-hez hozzáférhető operációs rendszer "port" beszerzését jelenti.) Az OS_CPU.H tartalmazza az adatok méretét, az alacsony szintű függvények definícióit, és a HAL (Hardware abstraction layer függvénycsomagot). A CPU_A.S file a legalapvetőbb négy függvény implementációját tartalmazza az adott processzorra. Ez assembly forráskód:

- megszakítás engedélyezés
- megszakítás tiltás
- a legmagasabb prioritású task kiválasztása egy várakozási sorból
- kontextus váltás

Az OS_CPU_C.C file tartalmazza az un. Hook függvényeket, amelyeket a tervező kitölthet egyedi műveletekkel, és ezeket a kernel automatikusan meghívja kontextus váltáskor, időzítő megszakítás kiszolgálásakor stb. Itt találjuk a taskok inicializálását, a hardware kezdeti állapotának beállítását és egyéb kezdeti műveleteket, amelyekhez nem kell közvetlen regiszter művelet.

A mélyen beágyazott rendszerekben, ahol a kernel csak a legalapvetőbb szolgáltatásokat nyújtja a legalsó réteggel egy szinten helyezkednek el a tervező által írott hardware kezelő és megszakítás kezelő függvények is. Ezeket a függvényeket a legfelső szintről hívjuk.