Diplomaterv

Gábriel Zoltán

2004.

BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
VILLAMOSMÉRNÖKI ÉS INFORMATIKAI KAR
MÉRÉSTECHNIKA ÉS INFORMÁCIÓS RENDSZEREK TANSZÉK

DIPLOMATERV FELADAT

Gábriel Zoltán

szigorló mérnök-informatikus hallgató részére
(nappali tagozat műszaki informatika szak)

Flexibilis futtatórendszer beágyazott információs rendszerekhez
(a feladat szövege a mellékletben)

A tervfeladatot összeállította és a tervfeladat tanszéki konzulense:

dr. Péceli Gábor
egyetemi tanár

A záróvizsga tárgyai:

Beágyazott inf. rendszerek
Deklaratív programozás
Kooperatív rendszerek

A tervfeladat kiadásának napja:
A tervfeladat beadásának határideje:

dr. Görgényi András
adjunktus, diplomaterv felelős

dr. Péceli Gábor
egyetemi tanár, tanszékvezető

A tervet bevette:
A terv beadásának dátuma:
A terv bírálója:

Nyilatkozat

Alulírott *Gábriel Zoltán*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Gábriel Zoltán
hallgató

Tartalomjegyzék

Kivonat

A diplomaterv egy valós idejű operációs rendszer megtervezésével és implementálásával foglalkozik.

A hagyományos valós idejű operációs rendszerek biztonsági megfontolásokból leginkább prioritás alapú, statikus ütemezőt használnak. Bizonyos alkalmazásoknál azonban felmerül az igény a külső körülményekhez jobban alkalmazkodó rendszerek iránt. Az ilyen rendszerek nem a futtatandó taszkok állandó prioritása alapján ütemeznek, hanem például a határidejük alapján. Így mindig az éppen legsürgősebb feladat jut processzor időhöz.

Előfordulhat azonban, hogy a rendszer még így is túlterhelt állapotba kerül. Ilyenkor elkerülhetetlen az adatvesztés, hiszen a határidejéig be nem fejeződött feladat összes addigi eredménye elveszik. Ez sok esetben megengedhetetlen. Erre a problémára nyújthat megoldást, ha a taszkokhoz többféle futási módot, feldolgozási szintet rendelünk, melyek mindegyike különböző futási idővel rendelkezik. Így az ütemező választhatja ki, hogy az éppen aktuális terheltségi szint mellett mekkora számítási kapacitás adható egy feladatnak. A csökkentett futási módban előállított eredmény természetesen nem lesz olyan pontos, mint a teljes értékű feldolgozás során kapott, azonban így mégis tudunk valamilyen eredménnyel szolgálni.

A megtervezendő és elkészítendő operációs rendszernek a fent bemutatott ütemezővel kell rendelkeznie. Ezen kívül biztosítania kell egy általános valós idejű operációs rendszer működéséhez szükséges szolgáltatásokat. Ilyen például az operációs rendszert vezérlő funkciók, a taszkok működését irányító funkciók. Szükséges biztosítani a taszkok közötti kommunikáció eszközeit. Az operációs rendszert továbbá fel kell készíteni egy külső monitorozó állomáshoz való csatolásra is, mely az operációs rendszer működésével kapcsolatos adatokat képes elemezni.

A funkciók kiválasztása után meg kell tervezni azok együttműködését, majd az így kapott rendszerterv alapján implementálni kell a rendszert. Ennek során hangsúlyt kell fektetni az operációs rendszer könnyű hordozhatóságára. A létrehozott rendszert a fejlesztés közben, és a végén is tesztelni kell, majd a következtetéseket levonva további fejlesztési célokat lehet kitűzni.

Abstract

This master thesis discusses the planning and implementation of a real-time operating system.

Traditional real-time operating systems – because of security considerations – use mainly static, priority based scheduling. There are some applications however, where the need for systems, which are more adaptive to their environments, arises. These systems do not schedule based on the constant priorities of the tasks, but for example based on the tasks' deadlines. This way it is always the most urgent task that gets processing time.

There are cases where even a dynamic system becomes overloaded. When a scenario like this arises, the loss of data is inevitable, because the task which did not finish its job until the deadline loses all its results. This cannot be allowed in many cases. A solution to this problem might be if we assign multiple modes of operation (quality of service levels) to a task. Each of these modes has a different computation time. This way the scheduler can choose which operating mode the task should use, based on the current system load. The result you get with the decreased quality level will certainly be less accurate than the one you would get with using the highest level. But at least we get a result.

The operating system to be planned and implemented should use the above mentioned scheduling algorithm. It also has to provide the usual services of a real-time operating system. This includes for example the functions to control the whole system and the task control functions. It also has to include methods that enable inter-task communications. Besides these requirements, the operating system needs to be able to handle an external monitoring station, which can analyze the data that describes the running of the operating system.

After selecting the services, their interconnections should be planned. Based on the system plan that has been conceived this way, the system should be implemented. During this process it should be ensured that the operating system stays easy to port to other platforms as well. The system should be tested in and after the development process. After drawing the conclusions new paths of development can be set.

Előszó

A diplomatervezési feladatom egy beágyazott környezetben alkalmazható valós idejű futtató rendszer specifikálása, megtervezése és implementálása.

A kereskedelmi forgalomban kapható, illetve szabadon elérhető beágyazott operációs rendszereknek számos fajtája létezik. Ezek általában különféle igények kiszolgálására íródnak. Megtalálhatóak köztük az egészen kis mikrokontrolleres rendszerektől kezdve a bonyolultabb, elosztott rendszereket is támogató megoldások. Ezen rendszerek nagy része számára azonban már fordítási időben eldől, hogy mikor milyen feladatot kell végrehajtaniuk. Így a környezet változásaira kevésbé tudnak reagálni.

Az általam megvalósítandó kísérleti operációs rendszernek éppen ezért rendelkeznie kell a futás közbeni adaptáció képességével. Vagyis képesnek kell lennie a feldolgozási szintek dinamikus változtatására az aktuális terheltségi szinttől függően. Ezt az ütemezési algoritmus megfelelő megválasztásával lehet elérni.

Mivel a rendszernek egy általános futtató rendszer szerepét kell ellátnia, ezért tartalmaznia kell az azokban megtalálható alapvető szolgáltatásokat is.

Az ütemezési algoritmus és a szolgáltatások kiválasztása után elkezdődhet a rendszer tervezése, majd a tervek alapján az implementálás.

Az implementálás során tesztelni kell a rendszer egyes komponenseit, majd a végén az egész rendszer együttműködését.

A diplomaterv egyes fejezetei a fenti folyamatot követik nyomon.

Az első fejezet egy általános bevezető a beágyazott, illetve valós idejű rendszerekkel kapcsolatban; azok legfőbb tulajdonságait ismerteti.

A második fejezet az ütemezésekről szól. Először az eddig alkalmazott statikus módszereket ismertetem, majd a legelterjedtebb dinamikus megoldásokat. Bemutatok egy adaptív megoldást is, majd végül az általam használandó ütemezési algoritmust.

A harmadik fejezet egy összehasonlítás néhány általam kiválasztott szabadon elérhető beágyazott rendszerről. A rendszereket a nyújtott szolgáltatásaik alapján hasonlítom össze. A szolgáltatásokat kategóriákba sorolva tárgyalom.

A negyedik fejezetben az előbb felsorolt szolgáltatásokból választom ki az operációs rendszerben felhasználandókat, majd a használt fejlesztési módszertant választom ki. Ezután, mivel a szükséges információk már megvannak, felállítok egy rendszertervet, statikus és dinamikus modellekkel.

Az ötödik fejezet a rendszerterv alapján megvalósított rendszerrel foglalkozik. A rendszer egyes osztályainak, metódusainak működését írja le sorban.

A hatodik fejezetben a tesztelési módszert ismertetem, majd a tesztelés eredményeit. Ezt követően összefoglalom a diplomatervezés eredményeit és tapasztalatait, végül pedig ismertetem a lehetséges továbbfejlesztési irányokat.

A diplomaterv első függeléke az operációs rendszer fordításával foglalkozik, az ezzel kapcsolatos tudnivalókat foglalja össze.

A második függelék az operációs rendszerhez írandó alkalmazások megvalósításához nyújt segítséget.

Végül a harmadik függelék a diplomaterv mellé benyújtott CD tartalmát ismerteti.

1. fejezet

Bevezetés

Beágyazott rendszereket az élet legkülönfélébb területein használnak. Ezek közös jellemzője, hogy általában egy adott környezetbe beágyazva működnek, és előre meghatározott feladatot hajtanak végre, melyhez a környezetből nyernek ki információt. Találhatunk ilyen rendszereket például gépjárművekben, repülőgépekbe építve, ipari folyamatok érzékelőiként és beavatkozóiként, vagy akár szórakoztatóelektronikai termékekben.

A beágyazott rendszer kifejezés mellé gyakran társul a valós idejű jelző is, ezek már-már egymás szinonímáivá váltak. A valós idejűség arra utal, hogy a rendszernek időben követnie kell a környezetét, azzal szinkronban kell együttműködnie.

A valós idejűség tehát nem egyenlő a rendszer gyors működésével, hanem sokkal inkább azt fejezi ki, hogy amit végrehajt, azt bizonyos időkorlátokon belül teszi. Vegyünk például egy olyan ipari folyamatot, melyben csokoládé szeleteket szeretnénk csomagolni. Az elkészült szeletek futószalagon érkeznek, és a rendszernek ezeket kell tudnia helyesen becsomagolni. Ehhez szükséges, hogy a rendszer szinkronban legyen a futószalaggal, és mindig a megfelelő időben hajtsa végre a csomagolási folyamatot. Két csokoládé szelet érkezése között a számítógép számára nagy idő telik el, tehát nem a gyorsaság a fő szempont, hanem az, hogy garanciát tudjunk vállalni arra, hogy a csomagolás adott időben hajtódjon végre. A rendszer ezt többek közt a folyamatokhoz rendelt határidőkel tudja elérni.

Beszélhetünk kemény- és puha valós idejű rendszerekről is. A puha valós idejű rendszerek olyan rendszerek, melyeknél egy határidő túllépése nem okoz komoly problémát. Itt megengedhető, hogy egy folyamat a határideje után is fusson, az ütemező mindössze igyekszik a követelményeket betartani.

Kemény valós idejű rendszerek esetén azonban egy határidő megsértése nagyon komoly következményekkel is járhat. Gondoljunk például egy repülőgépes rendszerre, ahol a helyes működésen akár emberéletek is múlhatnak. Az ilyen rendszereknél határidő túllépése esetén minimalizálni kell az esetleges kárt. A határidőt túllépő folyamatot általában le kell állítani.

A beágyazott rendszerek felépítése az alkalmazási környezet és a feladatorientáltság miatt eltér a megszokott asztali számítógépekétől. Egy beágyazott rendszer esetében az alkalmazásokat általában együtt fordítjuk az operációs rendszerrel, így a kettő egy egységet alkot. Ugyanez az asztali rendszerekre nem jellemző, ott a

kettő elkülönül egymástól. Beágyazott rendszerek esetében az alkalmazás indítja az operációs rendszert, míg asztali esetben pont fordítva. Egy beágyazott rendszernél a taszkok védelmére is kevesebb figyelmet kell fordítani, hiszen azok működése már fordítási időben eldől, utána nem változik. Jellemző még a beágyazott operációs rendszerek nagyfokú skálázhatósága. Fordítási időben eldönthetjük, hogy a rendszer mely komponenseit szeretnénk majd használni, és a lefordított kódba majd csak ezek fognak bekerülni.

Beágyazott rendszerek esetében gyakran jellemző az erőforrás szegény környezet. Ez általában lassabb processzort, kisebb rendelkezésre álló memóriát jelent. Ezért ezeknél a rendszereknél az egyik legnehezebb feladat az, hogy a meglévő erőforrások mellett a lehető legnagyobb teljesítményt tudják nyújtani. A memória felhasználás azonban csak a számítási teljesítmény rovására csökkenthető, és fordítva. A számítási teljesítmény növelése is csak több felhasznált memóriával érhető el. A két feltétel tehát egymásnak ellentmond, így nagyon nehéz a kettő közti optimális egyensúlyi állapotot megtalálni. Ez a kihívás jelenti a beágyazott rendszerek tervezésének egyik legnagyobb nehézségét, ugyanakkor a szépségét is. Asztali rendszereknél ugyanis az egyre nagyobb mennyiségben rendelkezésre álló erőforrások miatt ma már alig van szükség a teljesítmény ilyen mértékű optimalizálására.

2. fejezet

Ütemezési algoritmusok

Ebben a fejezetben a beágyazott, valós idejű rendszerekben leginkább alkalmazott ütemezési algoritmusokat mutatom be. Először a statikusakat, a dinamikusakat, majd a kettő ötvözéséből álló hibrid módszert. Ezt követően egy dinamikus, adaptív módszert mutatok be, majd végül az általam alkalmazandó algoritmust.

Az ütemezési algoritmusok adják meg az egyik legfontosabb különbséget két beágyazott rendszer között. Segítségükkel tudjuk szabályozni a beágyazott rendszer erőforrás felhasználását. Egy megfelelő algoritmussal növelhető a processzor kihasználtsága, vagy éppen csökkenthető a felhasznált memória mennyisége. Ezen a téren tehát többféle stratégia létezik. Egy egyszerű algoritmus például nem feltétlenül tudja kihasználni a processzor teljes számítási kapacitását, viszont mégis sikeres lehet, mivel kevés erőforrást fordít magára az ütemezésre. Egy bonyolultabb algoritmus ugyanakkor akár teljes mértékben ki tudja használni a processzort, azonban annyi erőforrást használ ehhez fel, hogy mégsem lesz jobb az egyszerűbb változatnál. A jól kiválasztott ütemezési algoritmus tehát nagy mértékben befolyásolja a rendszer teljesítményét.

2.1. Statikus algoritmusok

Ezen algoritmusok fő jellemzője, hogy a taszkokat előre meghatározott, statikus tulajdonságok alapján ütemezik, melyek később a futás során már nem változnak.

Az egyik ilyen típusú ütemezés a statikus prioritás alapú ütemezés. Itt a taszkok előre meghatározott prioritással rendelkeznek, és mindig a legnagyobb prioritású futásra kész taszkot futtatja az ütemező. Ebben a rendszerben tehát a processzort csak akkor vehetik el egy taszktól, ha az lemond róla, vagy egy nagyobb prioritású taszk lesz futásra kész.

Egy másik ilyen típusú algoritmus a *Rate Monotonic* ütemezés [Buttazzo02]. Az ütemezés periodikus taszkokkal dolgozik. A taszkokhoz az ütemező a periódusuk alapján rendel prioritást. Minél kisebb egy taszk periódusideje, vagyis minél nagyobb a gyakorisága, annál nagyobb lesz a prioritása. Megmutatható, hogy ilyen ütemező

használatával minden taszk ütemezhető lesz, ha azok összes processzor-kihasználása nem haladja meg a 69%-ot.

2.2. Dinamikus algoritmusok

Ezeknél az algoritmusoknál a feltétel, amely alapján a taszkokat ütemezi a rendszer, időben dinamikusan változik. Ezen algoritmusok erőforrásigénye nagyobb, mint az előzőekben bemutatatott statikus algoritmusoké, azonban segítségükkel nagyobb processzorkihasználtság esetén is ütemezhetőek lesznek a taszkok.

A legelterjedtebb dinamikus ütemezési algoritmus az *Earliest Deadline First* (EDF, legkorábbi határidejű először) [Buttazzo02], amelynél mindig az a taszk fog futni, amelyiknek a legközelebb van a határideje az aktuális rendszeridőhöz. Itt tehát a határidő alapján ütemezzük a taszkokat. Megmutatható, hogy az EDF algoritmus a taszkok 100%-os processzor-kihasználása mellett is ütemezni tudja azokat. Cserébe azonban bonyolultabb implementálni, hiszen minden ütemezésnél el kell dönteni, hogy éppen melyik taszknak van közelebb a határideje. Az algoritmus használata emellett több taszkváltással is jár, mint a statikus esetben. Látható tehát, hogy a teljesítmény növekedéséért számítási idővel fizettünk.

Egy másik hasonló filozófiát követő algoritmus a *Least Laxity First* (LLF, legkisebb hátralevő idejű először), amelynél a taszk határidejéig hátralevő idők alapján ütemezünk. Ezzel ugyanakkora kihasználtságot érhetünk el, mint az EDF ütemezővel, azonban az implementálása még költségesebb, hiszen a hátralevő idő a taszk aktuális futása alatt is változik, míg a határidő nem. Így tehát nő az adminisztrációs költség, valamint sűrűbbek lesznek a taszkváltások is.

2.3. Hibrid algoritmusok

A fenti két módszer ötvözésével kapjuk a hibrid ütemezési módszereket. Ezzel a módszerrel a valós- és nem valós idejű taszkok együttes ütemezését lehet hatékonyan kézben tartani. Az egyik ilyen módszer a *Maximum Urgency First* (MUF, legsürgősebb először)[GKSC02], ahol a taszkok különböző prioritási szinttel rendelkeznek, és az egy szinten levők közt LLF ütemezőt használ a rendszer. A másik ilyen módszer az *RMS+MLF* (Rate Monotonic Scheduling + Minimum Laxity First)[GKSC02]. Itt a kritikus taszkok ütemezéséhez a statikus Rate Monotonic algoritmust használják, míg az alacsonyabb prioritásúakhoz a dinamikus LLF ütemezőt.

2.4. Adaptív algoritmusok

Az adaptív ütemezési algoritmusok annyiban térnek el az előzőektől, hogy visszacsatolást biztosítanak a taszkok felé. Azaz tudják állítani a taszkok bizonyos futási

paramétereit, így javítva az egész rendszer teljesítményét. Periodikus taszkoknál például a periódusidő változtatásával csökkenthetjük, illetve növelhetjük egy taszk erőforrás használatát. Erre a módszerre alapul a következő alfejezetben részletesebben bemutatott módszer, valamint a az alábbi írásban bemutatott algoritmus: [DVJGS99].

A másik módszer a taszk által végrehajtott feladat minőségének befolyásolása. A gyengébb minőség ugyanis kisebb feldolgozási időt jelent. Egy képfeldolgozó rendszer esetében például lehet kisebb méretű, vagy kevésbé feldolgozott képekkel dolgozni [ZiLoSh02]. Vagy egy videofelvételekkel dolgozó alkalmazásban csökkenthető a képkockák részletessége, illetve a feldolgozott képkockák másodpercenkénti gyakorisága is [NBGG02][CJCP00]. Hasonló megközelítést alkalmaz az általam megvalósítandó rendszer is, azzal a különbséggel, hogy míg az előző megoldások egy már létező operációs rendszer felett, a középrétegben oldották meg az adaptív ütemezést, addig az én esetemben az ezt megvalósító algoritmus közvetlenül az operációs rendszer része lesz.

2.4.1. Egy dinamikus, adaptív algoritmus

Az algoritmust Giorgio Buttazzo és Luca Abeni javasolta, a neve elasztikus ütemezés (*Elastic Scheduling*) [BuAb02][SLCB00][GLCA02]. Az ütemezési algoritmus a dinamikus EDF ütemezőre épül, és periodikus taszkokat tud kezelni. A taszkok periódusidejeit változtatva éri el, hogy az adott terheltségi szint mellett minden taszk le tudjon futni.

Ehhez az algoritmus a következő adatokat tudja egy taszkról: a legrosszabb esetben előforduló (*worst-case*) számítási idő egy felső becslése; minimum, vagy névleges periódusidő; maximum periódusidő, valamint egy elasztikus együttható. Az algoritmus feltételezi, hogy a számítási idők előre nem ismertek. Azokat egy monitorozó mechanizmus figyeli futás közben, és mond becslést az értékükre. Az algoritmus a taszkokat rugókként kezeli, ahol a minimum periódusidő felel meg a rugó névleges hosszának, a maximum periódusidő a rugó teljesen összenyomott állapotának, az elasztikus együttható pedig a rugóegyütthatónak. Ha a taszkokat jelképező rugókat egymás mellé tesszük, kinyújtott állapotban, akkor az így kapott hossz felel meg a taszkok alapértelmezett állapotának. Ha adott a hosszak összegére egy korlát, amit a rugók túllépnek, akkor kell találni egy olyan F erőt, amivel az egész rendszert összenyomva a korlát alá kerülnek a rugóink. Egy ilyen F erő hatására minden rugó pontosan annyit megy össze, amennyire a rugóegyütthatója, illetve a minimális hossza engedi.

Ugyanezt a taszkok szemszögéből nézve hasonló eredményre juthatunk. Ott a korlát az alkalmazott ütemezési algoritmus ütemezhetőségi korlátja. Mint azt az előzőekben említettem, az EDF esetében ez 1, azaz a processzor 100%-os kihasználtsága mellett a taszkok még ütemezhetőek. A rugók hosszának most a taszkok processzor-kihasználása felel meg, azaz a számítási és a periódusidő hányadosa. Ha ennek az összértéke meghaladja az 1-et, akkor a taszkot periódusát meg kell növelni.

Azt pedig, hogy melyik taszknál mennyivel, az elasztikus együttható, valamint a maximum periódus határozza meg.

Az algoritmus nagy előnye, hogy a taszkok számítási ideinek előzetes ismerete nem szükséges, azt az algoritmus futás közben meghatározza.

2.4.2. Az általam használandó módszer

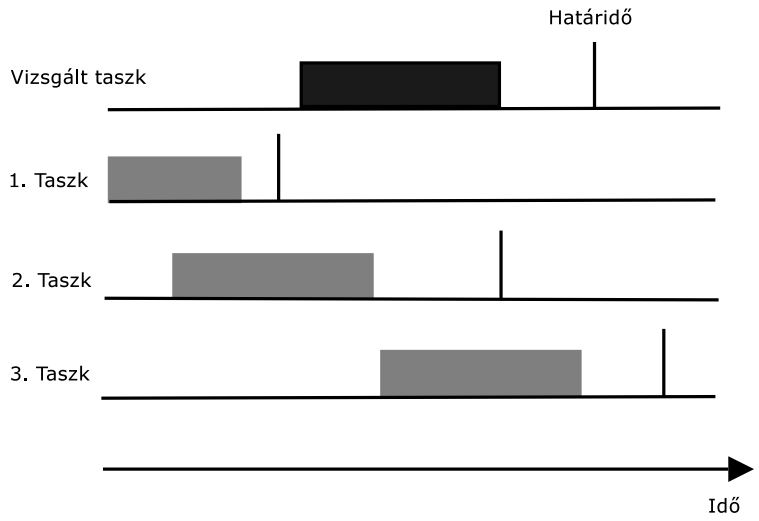
Az általam használandó módszer szintén az EDF ütemezőre épül. Ebben a felfogásban nem a taszkok periódusidejét változtatjuk a terheléstől függően, hanem a számítási idejét. Ezt úgy érhetjük el, hogy a taszkokhoz többféle futási módot rendelünk, melyek mindegyike egy-egy minőségi szintet (Quality of Service) testesít meg. Az algoritmus előre ismeri a taszkok periódusidejét, határidejét, valamint az egyes futási módokhoz tartozó számítási idők becslését.

Az algoritmus egy táblázatot tart nyílván, amiben a legközelebb futó néhány taszk fontos adatai találhatóak. Ezek a következők:

- Taszk azonosító
- Futási mód azonosítója
- Futási módok számítási idei
- A taszk határidejéig fontos számítási idők összege
- A taszk határideje
- A taszk indulási ideje

A fontos számítási idők összege azon taszkok számítási idejeiből tevődik össze, melyek határideje kisebb a vizsgált taszk határidejénél, továbbá nagyobb a vizsgált taszk indulási idejénél. Ezt ábrázolja a 2.1. ábra, ahol csak a 2. taszk fog bekerülni az összegbe. Az algoritmus minden taszk számára kiszámolja ezt az értéket, amikor az bekerül a táblázatba. Ezután megnézi, hogy a taszk határidejéig mennyi szabad számítási idő marad (ehhez az összegzett számítási időket veszi alapul), és ez a taszk melyik futási módjára elég. Ha a taszk egyik módjával se fér be az ütemezésbe, akkor az algoritmus végigmegy a táblázat azon taszkjain, melyek bekerültek a számítási összegbe, és ezek számítási idejét próbálja meg csökkenteni. Ha talál egy taszkot, amelyet ily módon csökkentve az új taszk futtatható, akkor véglegesíti a változást, és az új taszkot is felveszi a legkisebb futási móddal.

Ha egy taszk elkezdi futását, akkor kikerül a táblázatból, és a táblázat eltolódik, a végére pedig a még nem szereplő taszkokból kerül a legközelebb futó. Ha egy taszk nem található a táblázatban, de futni szeretne, akkor az algoritmus a számára ugyanígy elkészít egy számítási idő összeget, ami alapján eldönthető, hogy a taszk milyen módban kezdje a futást. Itt azonban ha a taszk nem futtatható, akkor az algoritmus már nem próbálja meg a többi taszkot csökkenteni.



2.1. ábra. A számítási idők összegének meghatározása

Az algoritmus előnye az előbb ismertetett elasztikus ütemezéssel szemben, hogy olyan esetekben is alkalmazható, amikor a szóban forgó rendszer fázisérzékeny, hiszen ilyenkor a periódusidők eltolása nem megengedhető. Az algoritmus továbbá képes az aperiodikus taszkok kezelésére is.

Az algoritmus hátránya, hogy a számítási időket előre meg kell becsülni hozzá. Ezen valamelyest csökkenthet egy külső monitorozó alkalmazása [Bartha04], amely figyeli a taszkok tényleges futási ideit, és azzal módosítja a taszkokat leíró struktúrákat. Az algoritmus hátránya továbbá, hogy a taszkok futási módjainak megválasztására nincsen egy olyan egyszerűen alkalmazható algoritmus, mint amely a rugós hasonlat volt az előző esetben.

3. fejezet

Operációs rendszerek összehasonlítása

A következőkben több valós idejű operációs rendszer szolgáltatásait gyűjtöm és hasonlítom össze, hogy képet kaphassak az azokban található megoldásokról és ezekre alapozva összeállíthassak a megtervezendő rendszer számára egy szolgáltatáslistát. Az összehasonlításhoz a *Google* internetes kereső könyvtárában található nyílt forráskódú, valós idejű operációs rendszereket vettem alapul [Google], kiegészítve őket az általam már korábban megismert μC/OS (szintén forráskódban rendelkezésre álló) rendszerrel. A nyílt forráskódú rendszereknek megvan az az előnye, hogy a szolgáltatásaikat közvetlenül a forráskódban meg lehet nézni, így nem kell pusztán a hangzatos címszavakra hagyatkozni, mint az a kereskedelmi szoftvereknél gyakran előfordul. A kereskedelmi szoftverek gyártóin továbbá nagy a nyomás, hogy jól ellenőrzött, minősített szoftvereket adjanak ki. Így a szoftvercégek a termékeikben – érthető módon – hajlamosak a régi jól bevált megoldásokat alkalmazni, míg a nyílt forráskódú rendszereknél ez nem feltétlenül van így, több az új, kísérleti jellegű megoldás. Az utóbbiak tehát jellegükben jobban hasonlítanak az általam megtervezendő rendszerre.

Az operációs rendszereket hat szempont szerint vizsgálom:

- Rendszerrel kapcsolatos feladatok
- Taszkkezelés
- Taszkok közti kommunikáció
- Monitorozást elősegítő szolgáltatások
- Hordozhatóság, konfigurálhatóság
- Egyéb megoldások

Az első három szempont a minden operációs rendszer számára szükséges alapszolgáltatásokkal foglalkozik. Magukba foglalják az ütemezéssel, taszkváltással taszkkezeléssel és taszkok közti kommunikációval kapcsolatos megoldásokat, melyek nélkül

egyetlen operációs rendszer sem lenne használható. Az első csoport a legalapvetőbb feladatokat foglalja egybe, az operációs rendszer működtetését végző szolgáltatások találhatóak benne. A második csoport a taszkok adminisztratív feladataival foglalkozik. A harmadik pedig a taszkok együttműködését segítő lehetőségeket egyesíti.

A monitorozhatóság kérdését külön vizsgálom. Az ezt támogató szolgáltatások például a különböző statisztikákat készítő taszkok, valamint a hálózati kommunikációt megvalósító megoldások.

Mint az már kiderült, fontos szempont a hordozhatóság, illetve a konfigurálhatóság, ezért az operációs rendszerek ilyen képességeit is külön vizsgálom.

Az utolsó csoportba a máshova nem sorolható egyedi megoldások kerültek.

A vizsgálat során az alábbi operációs rendszereket vettem figyelembe:

μC/OS (1.11-es verzió) [uCOS] Egyszerű, jól felépített, könnyen testreszabható rendszer. Elsősorban beágyazott rendszerekben alkalmazzák. Megkapta a repülőgépipar DO-178B minősítését, jelezvén hogy biztonságkritikus környezetben is alkalmazható. A korábbi tanulmányaim során részletesen megismertem ezt a rendszert, ezért az összehasonlításban nagy súllyal fog szerepelni.

Hartik [Hartik] Kísérleti operációs rendszer. Dinamikus ütemezőt használ, ezért az én szempontomból különös jelentőséggel bír. A beágyazott rendszerekben való alkalmazás itt kevésbé hangsúlyos, leginkább az x86 architektúrát támogatja. Multimédia és irányítási alkalmazásokhoz fejlesztették ki. Szintén részletesebben fogom vizsgálni.

S.Ha.R.K. [SHaRK] Az előző Hartik rendszer továbbfejlesztése. A készítők a modularitásra helyezték a hangsúlyt. Az operációs rendszer különböző modulokból épül fel, melyek közül a fordítás során az alkalmazásfejlesztő válogathat. Az ütemezési stratégiák is modulokban találhatóak, így akár többféle ütemezőt is használhatunk. Ez az operációs rendszer céljaiban már nem hasonlít annyira a kitűzött feladathoz, ezért nem is vizsgálom olyan részletesen.

proc [proc] Kis méretű, beágyazott rendszerekben használható operációs rendszer. Moduláris felépítésű, prioritásos ütemezőt használ.

EROS [Eros] Nagy méretű operációs rendszer, amit inkább bonyolultabb alkalmazások futtatására terveztek. Nagy hangsúlyt fektettek a biztonságra és az erőforrás-igénylések kézbentartására. Mérete és bonyolultsága miatt csak kevéssé felel meg a célként kitűzött rendszer képének, ezért nem is fog nagy súllyal szerepelni az összehasonlításban.

Fiasco [Fiasco] Abból a célból fejlesztették ki, hogy a DROPS nevű operációs rendszer [DROPS] kernele legyen. Mérete elég nagy, ezért inkább nagyobb méretű beágyazott alkalmazások alapjaként szolgálhat. Jellegében szintén jelentősen eltér a megcélzottól.

RTEMS [RTEMS] Leginkább beágyazott rendszerekben való alkalmazásra tervezték, ám komplexitása jóval meghaladja például a μC/OS-ét. Fő jellegzetessége, hogy képes multiprocesszoros rendszereket is kezelni. Prioritásos ütemezővel rendelkezik, több szabványnak is megfelel (POSIX, ANSI C).

rtmk [rtmk] Egy egyszerű szerver működtetésére kifejlesztett kisméretű operációs rendszer. A taszkok szálakban futnak. Alapértelmezett esetben időosztásos ütemezőt használ, de lehetőség van a prioritásos ütemezésre való áttérésre.

TinyOS [Tiny] Egymással hálózati kapcsolatban levő érzékelők működtetésére fejlesztették ki. Ezek az érzékelők autonóm működésűek és rendkívül kevés erőforrással rendelkeznek. Az alkalmazási környezete elég specifikus, ezért a megvalósítása is nagyban eltér a fenti rendszerektől.

A fenti felsorolásból a diplomatervezés során megvalósítandó feladathoz a legközelebb a μC/OS, a proc és a Hartik rendszerek állnak. Előbbi kettő leginkább az egyszerűsége és a felépítése miatt, míg az utóbbi amiatt, hogy a vizsgált rendszerek közül az egyetlen olyan operációs rendszer (kivéve az utódját, a S.Ha.R.K.-ot), amely dinamikus ütemezőt használ. A megvalósítandó rendszernek tulajdonképpen ezen tulajdonságok ötvözéséből kellene előállnia, kiegészítve azokat a futás közbeni adaptivitás képességével.

Az összehasonlítás során a μC/OS-t és a Hartikot a fent említett hasonlóság miatt fogom részletesen vizsgálni. Az RTEMS operációs rendszert mint egy komplexebb rendszert, a TinyOS-t pedig az eltérő koncepciója miatt fogom összehasonlítani az előbbi kettővel. A többi operációs rendszer vizsgálata sem haszontalan, ám azokat kevésbé részletesen fogom tárgyalni.

3.1. Rendszerrel-kapcsolatos szolgáltatások

Ebbe a kategóriába sorolom a rendszerindító és -leállító függvényeket, az ütemezőt, a memóriakezelést és taszkváltást elősegítő hívásokat, valamint az idő kezelését lebonyolító eljárásokat.

A 3.1. táblázatban soroltam fel a négy részletesen vizsgált operációs rendszer rendszerrel kapcsolatos fontosabb szolgáltatásait.

Ebből kiderül, hogy a rendszer indítását mind a négy operációs rendszer támogatja, a leállítást viszont a μC/OS nem. A rendszer leállítása kétségkívül elegáns megoldás, azonban nem feltétlenül szükséges, hiszen egy beágyazott rendszer esetében feltehetjük, hogy az hosszú ideig fog autonóm módon működni anélkül, hogy le kellene állítani.

A μC/OS ütemezője a taszkok prioritása alapján dönti el, hogy melyik fusson legközelebb, míg a Hartik esetében ez dinamikusan változik, a taszkok határidejének függvényében. Az RTEMS szintén prioritásos ütemezőt használ. A TinyOS nagyon kilóg a többi közül az ütemezés és taszkkezelés szempontjából. A rendszernek saját programozási nyelve van, a *nesC*, ami nyelvi szinten támogatja a taszkokat. Az

3.1. táblázat. A négy kiválasztott kernel rendszerrel-kapcsolatos szolgáltatásai

Szolgáltatás	*µC/OS*	*Hartik*	*RTEMS*	*TinyOS*
Rendszer indítás	+	+	+	+
Rendszer leállítás	–	+	+	+
Ütemező	prioritásos	EDF	prioritásos	FIFO
Memóriafoglalás virtuális géppel	–	+	+	+
Taszkváltás virtuális géppel	+	+	+	+
Rendszeridő lekérdezése	+	+	+	+
Órafelbontás állítása	–	+	–	+

ütemező a FIFO elv alapján az először beérkezett taszkot futtatja. Az általam megoldandó feladathoz dinamikus ütemező használata szükséges.

A virtuális gép használata azért előnyös, mert ezzel el lehet takarni a konkrét hardver réteget a kernel elől. Így egyszerűbb lesz az operációs rendszer portolása, hiszen csak a virtuális gépet kell újraírni minden használni kívánt platformra. A taszkváltást mindegyik operációs rendszer ezzel a módszerrel oldja meg. A memóriakezeléshez a Hartik, az RTEMS és a TinyOS tisztán virtuális gépet használ, míg a μC/OS csak részben. Az utóbbinál ugyanis a memóriafoglalás a taszkot létrehozó függvényben van implementálva, ami ugyan külön, a hardverfüggő részben található, de mégsem különül el annyira élesen.

A rendszeridő lekérdezése mindegyik esetben megtalálható. Az óra felbontásának állítását a Hartik-nál és a TinyOS-nél beépített függvény támogatja, míg a másik két esetben ezt nekünk kell megírnunk.

A *S.Ha.R.K.* kernel a Hartik továbbfejlesztése, ezért annak összes szolgáltatása megtalálható benne. A létrehozásakor többek közt a modularitás volt a cél, így például többféle ütemező közül választhatunk (akár prioritásos, akár dinamikus).

A *proc* operációs rendszer viszonylag egyszerű, a μC/OS 1.11-hez hasonlóan kevés szolgáltatást nyújt. Nem támogatja a rendszer leállítását, prioritásos ütemezőt használ. A virtuális gép koncepció azonban itt is megjelenik, és az a Hartik kernelhez hasonlóan jól elkülöníthető az operációs rendszertől.

A *Fiasco* rendszer elég komplex és nagyméretű. Az ütemezője prioritásos, és a virtuális gépet használó megoldás itt is megtalálható.

Az *rtmk* rendszer a taszkokat szálakban futtatja, négy lehetséges ütemezési stratégiát használva: round robin, FIFO, időosztásos és prioritásos. A virtuális gép itt is megtalálható.

3.2. Taszkkezelő szolgáltatások

Olyan függvények tartoznak ide, mint a taszkok létrehozása, felfüggesztése, törlése. Itt részletezem továbbá a taszkok operációs rendszer által kezelt paramétereit is. Az összehasonlítás eredménye a 3.2. táblázatban látható.

3.2. táblázat. A négy kiválasztott operációs rendszer taszkkezelő szolgáltatásai

Szolgáltatás	*μC/OS*	*Hartik*	*RTEMS*	*TinyOS*
Taszk létrehozása	+	+	+	+
Taszk törlése	+	+	+	–
Taszk felfüggesztése	+	+	+	–
Taszk újraindítása	–	–	+	–
Taszk késleltetése	+	+	+	(+)
Taszk határidők figyelembevétele	–	+	+	–
Periodikus taszkok kezelése	–	+	+	(+)
Periódus félbeszakítása / kihagyása	–	–	+	–
Taszkok csoportokba szervezése	–	+	–	–
Futás közbeni paraméter-változtatás	+	+	+	–

A taszkok létrehozása olyan alapszolgáltatás, melynek minden operációs rendszerben jelen kell lennie. Kiemelendő, hogy a TinyOS-t kivéve futás közben is létrehozhatnak a taszkok más taszkokat. Ugyanígy megtalálható a törlés, a felfüggesztés és a késleletetés is a három másik esetben.

Mivel a Hartik EDF ütemezőt használ, ezért a taszkok határidejét figyelembe kell vennie. Háromféle taszkot tud kezelni, kemény valós idejűt, puha valós idejűt és nem valós idejűt. Ha egy kemény valós idejű taszk túllépi a határidejét, az egész rendszer leáll egy hibakóddal. Puha valós idejű esetben a taszk határideje arrébbtolódik, míg nem valós idejű esetben nem kell törődni azzal. A μC/OS prioritásos ütemezőt használ, aminek nincs szüksége a taszkok határidejére. Ha a felhasználó ilyen taszkokat szeretne létrehozni, akkor azok kezeléséről a taszkok kódjában saját magának kell gondoskodnia. Mindkét rendszerből hiányzik egy olyan mechanizmus, amely határidő túllépés esetén megszakítaná a futást. Az RTEMS azonban rendelkezik egy ilyen mechanizmussal. Továbbá képes a taszkok újraindítására is, ami azt jelenti, hogy a függvényhívás hatására a taszkok abba az állapotukba kerülnek, amelyben a létrehozásukkor voltak.

A Hartik kernelben lehetőség van csoportok létrehozására, így kényelmesen tudunk egyszerre több taszkot kezelni.

A periodikus taszkok kezelése csak a Hartik és az RTEMS esetében megoldott. A Hartik-nál a taszk kódjában elhelyezett függvényhívás jelzi az adott periódus végét. Az RTEMS a periodikus taszkokkal kapcsolatban képes a fent említett probléma kezelésére, mégpedig ha egy periódus túllépi a határidejét, akkor azt meg tudja szakítani, és ezután a következő periódus fog futni a megfelelő időben.

A Hartik különböző függvényhívásokkal támogatja a taszkok paramétereinek (határidő, periódusidő) futás közbeni megváltoztatását. Hasonló lehetőség a μC/OS-ben is található, de mivel az csak a taszkok prioritását kezeli, ezért csak azt lehet megváltoztatni. Az RTEMS nem tudja megváltoztatni futás közben egy taszk periódusát, határidejét.

A TinyOS a taszkok kezelése terén az előző három operációs rendszertől teljesen

eltérő filozófiát követ. A taszkok itt atomiak, olyan értelemben, hogy azok futását a megszakításokon kívül más nem szakíthatja félbe. A taszkoknak nincsenek paraméterei, és az ütemező FIFO sorrendben hajtja őket végre. A taszkok kezelését időzítőkkel oldhatjuk meg. Beállíthatunk például egy periodikus időzítőt, ami adott időközönként meghívja a taszkot. Vagy egy időzítő segítségével késleltethetjük is a taszk futását.

A taszkok kezelésében nincs nagy eltérés a többi operációs rendszer között. Az eddig nem tárgyalt rendszerek prioritásos ütemezőt használnak, így nem kezelik a határidőket és a periodikus taszkokat.

3.3. Kommunikációs szolgáltatások

Ebbe a csoportba a taszkok közötti információcserét segítő megoldások kerültek. A négy operációs rendszer ilyen szolgáltatásait a 3.3. táblázat foglalja össze.

A szemafor (az erőforrásokhoz való hozzáférésen kívül) a taszkok szinkronizálását segíti, és a TinyOS-en kívül mindhárom rendszerben megtalálható. A taszkok azzal, hogy a szemaforra várakoznak, bevárhatnak egy másik taszkot a működésében.

A sor a legáltalánosabb kommunikációs megoldás, hiszen mind a négy rendszer támogatja. Ez két taszk között egy FIFO-ként működve teszi lehetővé az üzenetek cseréjét. A postaláda segítségével pedig egy adott taszknak tud az összes többi üzenetet küldeni.

Ezeket az alapszolgáltatásokat a Hartik a státusz információkkal egészíti ki, amelybe csak egy taszk írhat, viszont az összes többi olvashatja. Az olvasás nem törli az üzenetet, tehát az mindig az író taszk legutolsó állapotát fejezi ki. Egy másik megoldás a ciklikus aszinkron puffer (*Cyclical Asynchronous Buffer – CAB*), ami nagyon hasonlít a státusz információkhoz. Annyi a különbség, hogy ebbe több taszk is írhat üzenetet, de ugyanúgy mindig az utoljára bekerült üzenetet lehet olvasni. Ezt a különböző periódusú taszkok együttműködésének megkönnyítésére találták ki. Ezeket a megoldásokat a többi rendszer nem alkalmazza.

Az RTEMS és a TinyOS támogatja ezen kívül az aszinkron üzenetküldést is, melyekkel a taszkok eseményeket és jelzéseket adhatnak át más taszkoknak. A fogadó taszkok az ilyen üzenetek olvasásához eseménykezelőket használnak, melyek leginkább a megszakítás-kezelőkre hasonlítanak. Amikor üzenet érkezik, ezek a jellemzően rövid függvények hívódnak meg, és kezelik le azokat.

A *proc* operációs rendszer csak a szemaforokat és sorokat valósítja meg, a postaládát nem. Sorból két fajtát ismer, az egyikkel bájtnyi információt lehet átadni (csatorna), a másikkal tetszőleges méretűt.

Az *EROS* operációs rendszerben a kommunikáció leginkább szolgáltatások kérésére szolgál. A szolgáltatásokat kulcsokkal lehet elérni. Egy üzenet négy kulcsot és egy lapnyi adatot tartalmazhat. Nagyobb méretű üzenet is átadható, ekkor egy memóriaterületet kell lefoglalni, és annak a kulcsát átküldeni.

3.3. táblázat. A négy kiválasztott operációs rendszer kommunikációs szolgáltatásai

Szolgáltatás	*µC/OS*	*Hartik*	*RTEMS*	*TinyOS*
Szemafor	+	+	+	–
Sor (Queue)	+	+	+	+
Többesküldés (Broadcast)	–	–	+	–
Postaláda (Mailbox)	+	+	(+)	–
Státusz információ	–	+	–	–
Ciklikus aszinkron puffer (CAB)	–	+	–	–
Események küldése	–	–	+	+
Jelzések küldése	–	–	+	+

3.4. táblázat. A négy kiválasztott operációs rendszer monitorozást elősegítő szolgáltatásai

Szolgáltatás	*µC/OS*	*Hartik*	*RTEMS*	*TinyOS*
Kommunikációs eszközök kezelése	–	+	+	+
Objektumok állapotának lekérdezése	+	+	+	+
Statisztika készítése	(+)	–	–	–
Eseménynapló vezetése	–	+	–	+

A *Fiasco* processzek közti hívásokkal (*Interprocess calls - IPC*) oldja meg a kommunikációt. Egy taszk küldhet üzenetet egy másiknak, illetve bármely másik taszktól várhat egy üzenetet.

Az *rtmk* rendszerben a kommunikáció portokon keresztül történik. A taszkok üzeneteket csak portnak küldhetnek, és csak azokból olvashatnak ki. A portokhoz való hozzáféréshez írási ill. olvasási jogosultsággal kell rendelkezni. Egy portba több taszk is írhat, de csak egy taszk olvashatja.

3.4. Monitorozást elősegítő szolgáltatások

Az operációs rendszer működéséről, belső állapotairól a teszteléshez, és a mindennapi működtetéshez is szükséges információkkal rendelkeznünk. Ezt megoldhatjuk utólagosan, ha például egy naplófájlba jegyezzük fel a történéseket. A rendszert monitorozhatjuk azonban folyamatosan, annak működése mellett. Ehhez használhatunk például egy külső számítógépet, ami hálózati kapcsolaton keresztül kapja az állapotinformációkat, majd azokat képes elemezni. Szerencsés, ha az operációs rendszer támogatja ezeket a megoldásokat. Az operációs rendszerek ilyen lehetőségeit a 3.4. táblázat foglalja össze.

A távoli monitorozáshoz az adatokat valahogy át kell juttatni a távoli számítógépre. Ehhez szükség van legalább egy hálózati eszköz támogatására. A µC/OS

3.5. táblázat. A négy kiválasztott operációs rendszer a hordozhatóság és konfigurálhatóság szempontjából

Szolgáltatás	*µC/OS*	*Hartik*	*RTEMS*	*TinyOS*
Processzorfüggő kódok elkülönítve	+	+	+	+
Linkelendő komponensek kiválaszthatók	+	+	+	+
Operációs rendszer paraméterezhető	+	+	+	–

nem rendelkezik ilyen képességgel. A Hartik UDP kapcsolatot lehetővé tevő Ethernet hálózati meghajtóval rendelkezik. Az RTEMS szintén az Ethernet hálózatot támogatja, rengeteg protokollt képes kezelni (például PPP, UDP, TCP). A TinyOS szenzorhálózatok működtetésére készült, melyek rádiós kapcsolatban állnak egymással, tehát támogatja a rádiós kapcsolatot.

A rendszer objektumainak állapotát mindegyik operációs rendszerben le lehet kérdezni. Ilyenek például a taszkok paraméterei, a sorok állapota, az ütemezésre váró taszkok száma.

A μC/OS 2-es verziója képes a futás közben statisztikát készíteni a processzor kihasználtságáról. Ezt egy statisztikai taszk futtatásával éri el, ami másodpercenként méri a terheltséget.

A Hartik rendszer képes az eseményeket egy naplóba rögzíteni. A futás során a napló a memóriában tárolódik, majd annak végeztével a rendszer kiírja egy fájlba. Az így kapott információkat később elemezhetjük. A TinyOS is képes naplózni a paramétereket. Ezek általában az érzékelő által mért értékek, melyeket egy idő után rádiós kapcsolaton keresztül elküld egy másik eszköznek.

A S.Ha.R.K. operációs rendszer már jóval több eszközt támogat a Hartik-nál, és a készítők szerint a rendszerrel együttműködik bármilyen szabad operációs rendszer meghajtó programja.

A proc és az rtmk az állapotlekérdezésen kívül nem támogatja egyik megoldást sem. Az EROS rendszer meghajtó-programokat nyújt Ethernet hálózati kártyákhoz, valamint képes az eseményeket naplózni. A Fiasco a soros porti kommunikációt támogatja.

3.5. Hordozhatóság, konfigurálhatóság

Beágyazott rendszereket nagyon sokféle környezetben használnak, melyekhez eltérő processzorok illetve mikrokontrollerek használata lehet ideális. Ezért nagyon fontos szempont, hogy az operációs rendszerek úgy épüljenek fel, hogy könnyen átültethetőek legyenek egy új platformra.

Az eltérő környezet miatti eltérő igények megkövetelik az operációs rendszerek testreszabhatóságát is. Előnyös, ha a felhasználó eldöntheti, hogy az adott feladat végrehajtásához milyen komponensekre van szüksége.

A 3.5. táblázat tartalmazza a négy operációs rendszer ilyen irányú tulajdonságait. A processzorfüggő kódok elkülönítése jól megoldja az egyszerű hordozhatóságot, hiszen csak ezeket kell módosítani egy új környezet esetén. A négy rendszer mindegyike így épül fel. A 3.1. fejezetben erről már volt szó a virtuális géppel kapcsolatban. Ott említettem, hogy a μC/OS nem teljesen átlátszó virtuális gépet használ, hiszen a memóriafoglalás a taszkot létrehozó függvényben szerepel (noha az a függvény szintén a processzorfüggő részben található).

A fordításnál a felhasználó mindegyik esetben eldöntheti, hogy melyik komponenseket szeretné használni. A μC/OS 1.11-es verziójában a komponensek egy fájlban találhatók, ezért fordítási direktívákkal oldották meg a kiválaszthatóságot. A többi esetben az egyes komponensek külön fájlokban találhatók, és az összeszerkesztésnél adhatjuk meg, melyikeket szeretnénk használni.

Az operációs rendszer konfigurálhatósága azt jelenti, hogy megadhatjuk például a használt sorok méretét, a taszkok maximális számát, az azoknak adható memória méretét. A vizsgált operációs rendszerek közül az első háromban ez lehetséges, a TinyOS-ben azonban nem találtam ilyen megoldást.

A processzorfüggő kódok elkülönítése általános megoldás, hiszen a többi rendszer is így épül fel. A linkelendő komponenseket a proc esetében nem lehet kiválasztani, az összes többi figyelembe vett rendszernél igen. A paraméterezhetőségre ugyanez igaz.

3.6. Egyéb szolgáltatások

A fent felsoroltakon kívül más, ezekbe a kategóriákba nem sorolható szolgáltatásokat is nyújtanak egyes operációs rendszerek.

A *Hartik* képes az egeret, a billentyűzetet és a képernyőt kezelni, utóbbit akár grafikus módban is. A S.Ha.R.K. ennél még több meghajtó-programmal rendelkezik.

Az *RTEMS* rendszer többféle fájlrendszert képes kezelni, valamint használhatjuk homogén ill. heterogén multiprocesszoros rendszerekben. A multiprocesszoros környezetet a rendszer automatikusan kezeli, a felhasználónak azzal nem kell törődnie.

Az *EROS* operációs rendszer egyedi megoldása, hogy képes a rendszerről szabályos időközönként „pillanatfelvételt” készíteni, így egy esetleges hiba esetén a rendszer pillanatokon belül helyre tud állni. Megemlítendő az is, hogy bizonyos kernel taszkok együtt futnak a felhasználói taszkokkal, így egy fontos taszk akár őket is meg tudja előzni.

4. fejezet

Rendszerterv

Az előző fejezetben az volt a célom, hogy egy általános képet kapjak a beágyazott operációs rendszerek által nyújtott szolgáltatásokról. A szolgáltatásokat csoportokra bontva vizsgáltam. Ebben a fejezetben az általam írandó operációs rendszerhez szükséges szolgáltatásokat fogom kiválasztani, majd ezek alapján felállítok egy előzetes rendszertervet. Az első alfejezet a kiválasztott szolgáltatásokat sorolja fel. A második alfejezetben az operációs rendszer fejlesztéséhez leginkább illő fejlesztési módszert választom ki. A harmadik alfejezet a rendszer implementálásához leginkább illő módszertan és környezet kiválasztásával foglalkozik. A negyedik alfejezet az előzetes rendszertervet tartalmazza, az első fele az objektum modellt, míg a második a dinamikus modelleket.

4.1. Kiválasztott szolgáltatások

Ebben az alfejezetben az előző, valós idejű operációs rendszereket összehasonlító fejezetnél látott szolgáltatások közül fogom kiválasztani a megtervezendő operációs rendszer szolgáltatásait.

4.1.1. Rendszerrel kapcsolatos szolgáltatások

A rendszerrel kapcsolatos szolgáltatások szempontjából nincs nagy eltérés a vizsgált operációs rendszerek között. Az itt felsorolt szolgáltatások nagy része szükséges egy operációs rendszer alapvető működéséhez. Az alábbi szolgáltatások implementálását tervezem az operációs rendszerbe:

- Rendszer indítása
- Rendszer leállítása
- EDF ütemező
- Memóriafoglalás virtuális géppel

- Taszkváltás virtuális géppel
- Rendszeridő lekérdezése

A *rendszer indítására* mindenképpen szükség van, hiszen e nélkül az operációs rendszer nem tudna működni. Ebbe a szolgáltatásba a rendszer ténylegesen indításán kívül beletartozik még az operációs rendszer belső változóinak inicializálása, az adatszerkezetek létrehozása és az üresjárati taszk létrehozása.

Mint azt az összehasonlítás során említettem, a *rendszer leállítása* funkció nem feltétlenül szükséges egy beágyazott rendszer számára, hiszen például egy irányítási alkalmazást akár évekig is hagyhatnak működni megállás nélkül. Azonban ez egy kétség kívül elegáns szolgáltatás, és az implementációs költsége sem túl nagy, így úgy döntöttem, hogy az operációs rendszer ezt is tartalmazni fogja.

A tervezendő operációs rendszer célja nagy mértékben az, hogy futás közben képes legyen adaptálódni a környezet aktuális elvárásaihoz. Ezt többek között *dinamikus ütemező* alkalmazásával éri el, amely a hagyományos prioritásos ütemezőknél jobb kihasználtságot eredményez. A választásom az EDF (*Earliest Deadline First*) ütemezőre esett, mely mindig a legközelebbi határidővel rendelkező taszkot futtatja. Egy másik lehetőség az LLF (*Least Laxity First*) ütemező lenne, amely a legkevesebb hátralevő idővel (a taszk határidejéig hátralevő idő) rendelkező taszkot futtatja. Azonban, mint azt az ütemezésről szóló fejezetben említettem, ez a sűrűbb taszkváltás miatt nagyobb terhet róna a processzorra.

A *memóriafoglalás és a taszkváltás* is fontos, nélkülözhetetlen feladat, mely nélkül az operációs rendszer nem lenne működőképes. Ezeket a funkciókat egy virtuális gép rétegként fogom megvalósítani, amely az operációs rendszer többi része elől eltakarja a tényleges hardvert.

A *rendszeridő lekérdezése* is szükséges, hiszen az operációs rendszernek tudnia kell, hogy például egy várakozó taszk mikor lehet ismét futásra kész. A szolgáltatás ezen kívül a taszkoknak is hasznos információt nyújthat.

A rendszeróra felbontásának állítására nem tervezek külön függvényt írni. A feladat szempontjából elég, ha azt fordítási időben állítani lehet a taszk forráskódjának átírásával.

4.1.2. Taszkkezelő szolgáltatások

A taszkokkal kapcsolatos feladatokat látják el. Az operációs rendszerek e szolgáltatások tekintetében már nagyobb különbséget mutatnak. A szolgáltatások meghatározásánál nagyobb szerepet kap az operációs rendszer tipikus alkalmazási köre. Az alábbi funkciókat tartom szükségesnek:

- Taszk létrehozása
- Taszk törlése
- Taszk felfüggesztése

- Taszk késleltetése
- Taszk határidők figyelembevétele
- Periodikus taszkok kezelése
- Periódus félbeszakítása
- Többféle végrehajtási mód kezelése

A *taszkok létrehozása* egy általános feladat, minden operációs rendszernél szükséges. A tervezendő operációs rendszernél fontos, hogy új taszkokat futás közben is létre lehessen hozni, ugyanis ezek jelképezhetik illetve kezelhetik a váratlan eseményeket.

A *taszkok törlésének* fontossága már nem ilyen egyértelmű, hiszen a feladatát befejező taszk számára elegendő lenne az is, ha felfüggesztett állapotba kerülne, úgy sem kapna processzoridőt. Figyelembe véve azonban a tényt, hogy beágyazott környezetre tervezek operációs rendszert, ez a funkció nem elhagyható. Ott ugyanis a rendszer a rendelkezésre álló memória szempontjából erősen korlátozott. Előfordulhatna tehát, hogy egy újonnan létrehozott taszknak a régi, már lefutottak miatt nem jut hely.

Egy *taszk felfüggesztése* többek közt akkor lehet hasznos, ha az valamiért elkezd rosszul működni, például kiesik a szinkronból. A *taszkok késleltetése* lehetőséget ad a fázisok módosítására. Ezt a feladatot csak operációs rendszer szinten lehet megoldani.

A futtató rendszer alapvetően a kemény valós idejű taszkokat fogja támogatni. Ez azt jelenti, hogy annak a *határidőket* is kezelni kell tudnia. A rendszer ezen kívül támogatja a *periodikus taszkokat* is, valamint képes lesz egy *periódus félbeszakítására*, ha az túllépi a határidejét.

A rendszer a futás közbeni adaptivitást többek közt úgy oldja meg, hogy a taszkok *többféle futási módot* kínálnak fel neki, melyek mindegyike különböző végrehajtási idővel rendelkezik. Az ütemező a működés során így mindig a helyzethez leginkább illő módot választhatja ki. Alacsony terhelés esetén a taszk a leghosszabb móddal futhat, nagy terhelés esetén viszont lehet, hogy már csak egy kisebbel. Ilyen funkció a többi operációs rendszerben nem található.

A taszkok újraindítására nincsen szükség, hiszen ha egy periodikus taszk lépi túl a határidőt, akkor az legközelebb a következő periódussal fogja folytatni a futást, ha pedig egy nem periodikus taszk teszi ezt, akkor az már nem fog legközelebb futni.

A taszkok csoportokba szervezése kényelmes funkció, azonban nem feltétlenül szükséges, így ezt a megoldást nem valósítom meg.

4.1.3. Kommunikációs szolgáltatások

Az alábbi kommunikációs szolgáltatások megvalósítását tervezem:

- Szemafor

- Sor
- Státusz információ

A *szemafor* és a *sor* a legalapvetőbb kommunikációs megoldás. Hasonlóan fontos még a postaláda, ám a sorokat használhatjuk postaládaként is, ha tervezési időben ügyelünk arra, hogy egy bizonyos sort csak egy taszk olvashasson. A *státusz információ* nem különbözik sokban a sortól, a különbség annyi, hogy ez utóbbi mérete egy, és az olvasás nem törli a tárolt üzenetet. Ez a megoldás tehát kevés plusz munkával megvalósítható.

Nagyon eltérő működésű az aszinkron kommunikáció (események és jelzések küldése), ami nyilvánvalóan nagyon hasznos megoldás, ám implementálása túlmutat a diplomatervezés keretein.

4.1.4. Monitorozás

A következő elemeket tervezem megvalósítani:

- Kommunikációs eszköz kezelése
- Objektumok állapotának lekérdezése
- Naplózás

Mint azt az összehasonlításnál már említettem, legalább egy *kommunikációs eszköz* támogatása szükséges a távoli monitorozáshoz, hiszen valahogy el kell tudni juttatni az adatokat az egyik gépről a másikra. Ez jelen esetben a soros porti adatcsere támogatását fogja jelenteni, melynek kódja a processzorfüggő részbe fog kerülni.

Az *objektumok állapotának* lekérdezése nem csak a monitorozásnál hasznos, előfordulhat, hogy egyes taszkok működésében is fontos szerepet játszik például egy sor állapota.

A *naplózás* a kommunikáció hatékonyságát növeli, ugyanis segítségével egy pufferben lehet gyűjteni az egyes eseményeket, majd egy adott idő után azokat egyszerre lehet elküldeni. Ez ugyan késlelteti az adatok beérkezését a monitorozó számítógépre, de a processzort nem lassítja az, hogy az adatokat minden esemény után rögtön el kell küldenie.

A *statisztika készítésének* implementálása szükségtelen, azt a külső monitorozó számítógép fogja a kapott adatok alapján elkészíteni.

4.1.5. Hordozhatóság, konfigurálhatóság

Mivel a hordozhatóság és a konfigurálhatóság fontos tulajdonság, ezért a többi operációs rendszernél előforduló három szempont mindegyikét alkalmazni fogom az implementálás során:

4.1. táblázat. A C nyelv előnyei és hátrányai

Előny	*Hátrány*
Könnyű hozzáférni a változókhoz	Kevésbé biztonságos
Kiforrott ezen a téren	Kevésbé strukturált
Gyors kód	

- Processzorfüggő kódok elkülönítve
- Linkelendő komponensek kiválaszthatóak
- Operációs rendszer paraméterezhető

4.2. Megközelítési mód

Programokat sok nyelven és sokféle fejlesztési módszertannal tervezhetünk illetve implementálhatunk. A beágyazott rendszerek esetében szinte egyeduralkodó a C nyelv használata. Előfordul még C++ (ilyen például a Fiasco) és assembly nyelvű megoldás is. A Java nyelv használata ezen a területen még nem mondható elterjedtnek, bár van erre is kezdeményezés, például a Real-Time Java [RTJava].

A négy programozási nyelv különböző tervezői hozzáállást igényel. C++ és Java esetén az objektum-orientált fejlesztési módszertant kell követnünk, míg a C nyelv esetében a hagyományos strukturált programozást. Az assembly nyelv esetében a megközelítés mód teljes mértékben processzor centrikus, a fejlesztést nagyban annak a képességei határozzák meg.

A négy nyelv közül kettő, a C és a C++ merül fel komoly alternatívaként. A teljes assembly nyelvű megvalósítás ugyanis nem teljesítené az egyszerű hordozhatóság követelményét, hiszen minden funkciót minden egyes új platform esetén újra kéne írni. A Java e területen meglevő kiforratlansága mellett hátránya az is, hogy a hardverkezelése elég nehézkes. A problémák ellenére azonban létezik Java alapú valós idejű operációs rendszer, az esmertec cég JBed [JBed] terméke, amely előremutató módon EDF ütemezőt alkalmaz.

Az alábbiakban a C és C++ nyelv előnyeit illetve hátrányait összefoglalva fogom kiválasztani a követelményekhez leginkább illő megoldást. A 4.1. táblázatban a C, míg a 4.2. táblázatban a C++ nyelv tulajdonságait soroltam fel.

A C nyelv legfőbb erénye a C++-szal szemben az, hogy lehetővé teszi az egyes változókhoz történő közvetlen hozzáférést. C++ használatakor ugyanis az egyes objektumok változóihoz általában csak a maguk az objektumok férhetnek hozzá, más objektumok pedig csak közvetítő függvényeken keresztül. Ezt azonban megkerülhetjük a C++-ban a *friend* mechanizmus segítségével, amely lehetőséget ad egyes barát osztályok számára, hogy közvetlenül hozzáférjenek az objektum változóihoz.

4.2. táblázat. A C++ nyelv előnyei és hátrányai

Előny	*Hátrány*
Biztonságosabb	Kevésbé kiforrott ezen a téren
Jobban strukturált	Objektumok meghatározása nehéz
Szabványos sablon könyvtár (STL)	Nehézkes hozzáférés a változókhoz
Szabványos modellek (UML)	Esetleg lassabb ill. nagyobb kód

A belső változók nehéz elérése azonban a biztonságot növeli, hiszen azokat kívülről csak a rendelkezésre bocsátott interfészeken keresztül fogjuk tudni elérni.

Az objektum-orientált programozási módszertan az adatokat és funkciókat objektumokba szervezi, melyek ezáltal egységes egészet alkotnak. Ez növeli a rendszer áttekinthetőségét és strukturáltságát. Jelen esetben azonban az objektumok meghatározása nem triviális feladat, hiszen az adatszerkezeteket (például a taszkok adatait) majd az összes funkció használja. Így könnyen mondhatnánk, hogy az egész operációs rendszer kerüljön egy objektumba.

A C++ nyelvvel kapcsolatban általános vélemény, hogy az abban írt kód lassabb lesz a C nyelvbeli megfelelőjénél. Ezt alátámasztja, hogy a vizsgált operációs rendszerek két kivétellel (a Fiasco és az RTEMS) mind C nyelven íródtak (illetve a TinyOS részben a saját nyelvében, a nesC-ben).

A C++ előnye a rendelkezésre álló szabványos sablon könyvtár (*Standard Template Library, STL*), amely sok adatszerkezetet valósít meg hatékony és hordozható módon, így az ezekre fordítandó munkát megspórolja.

Az objektum orientáltság mellett szólnak a szabványosított modellek is, mint az UML (*Unified Modeling Language*), mely a programok fejlesztését nagyban megkönnyíti.

A fent felsorolt érvek és ellenérvek alapján az operációs rendszert C++ nyelven fogom megvalósítani. A döntés során leginkább a strukturáltságot és a modellezési lehetőségeket vettem figyelembe.

Az operációs rendszer processzorfüggő komponenseit azonban assembly nyelven kell megvalósítani, hiszen ezek a feladatok mind hardverközeliek, és a megoldásuk bonyolultabb és kevésbé hatékony lenne mind C, mind C++ nyelven.

4.3. Rendszerterv

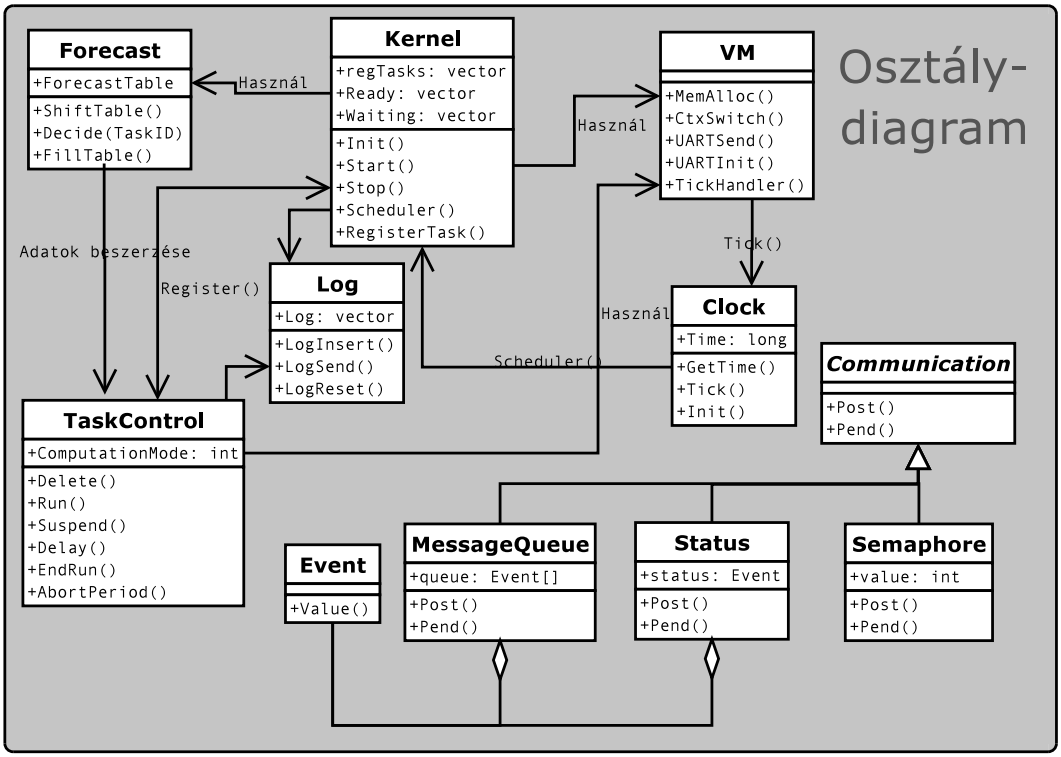
Mivel az objektum-orientált megközelítés mellett döntöttem, a legfontosabb feladat az osztályok megtervezése. Ehhez a megvalósítani kívánt szolgáltatásokból összefüggő, konzisztens csoportokat kell létrehozni. Az osztályok létrehozása után azok egymás közti kapcsolatait kell feltérképezni, illetve az együttműködésük módját.

Kondorosi, László és Szirmay-Kalos az Objektum-orientált szoftverfejlesztés című könyvük valós idejű rendszerekkel foglalkozó fejezetében [KLSz99] javaslatot tesznek az ilyen környezetben hatékonynak mondható fejlesztési módszertanra. En-

nek első lépéseként a rendszer és a környezet kapcsolatát célszerű feltárni. Ez a lépés azonban jelen esetben kihagyható, hiszen egy általános futtató rendszer tervezése a cél, ami bármely környezetben megállja a helyét, a külső kapcsolatok feltárása pedig inkább egy komplett alkalmazás fejlesztésekor hasznos. Az operációs rendszer taszkok felé nyújtott szolgáltatásait pedig éppen az előző fejezet tárta fel. Ezután a feladatot több párhuzamosan működő komponensre célszerű bontani, majd ezek dinamikus modelljét érdemes vizsgálni. A szerzők szerint a valós idejű rendszerek általában jobban megragadhatók a dinamikus modellekkel, mint a statikus leírásmódokkal, hiszen itt központi szerepet játszik az idő.

4.3.1. Statikus modell

Először tehát a kiválasztott szolgáltatásokat csoportosítottam úgy, hogy a csoportok egy egységes osztályt alkossanak. A létrehozott osztályokat és azok kapcsolatait a 4.1. ábra mutatja.



4.1. ábra. Osztály diagram

A *Kernel* osztályba az egész rendszerrel kapcsolatos funkciók kerültek, mint például az inicializálás, az operációs rendszer indítása és a leállítása. Szintén itt található az ütemező is, ami a taszkok határideje alapján választja ki, hogy melyik fusson. A Kernel osztály tartalmazza a futásra kész és várakozó taszkok listáját is, melyek a taszkok mutatóiból álló vektorok.

A *VM* osztály a virtuális gépet valósítja meg. Funkciói a memóriafoglalás, a taszkváltás, a soros porti kommunikáció kezelése (inicializálás és adatok küldése) és a hardveróra megszakítását lekezelő interrupt függvény.

A *TaskControl* osztály a taszkokért felelős absztrakt osztály. Itt tárolódnak a taszkok paraméterei (periódus, határidő, ofszet, számítási idők) és ez az osztály felelős a taszkokkal kapcsolatos műveletekért (törlés, futtatás, felfüggesztés, késleltetés, periódus befejezése). A regisztrálás a Kernel osztály felé szükséges, hogy az elhelyezhesse a taszkot a megfelelő listájában (futásra kész, várakozó). Az osztály mint már említettem a taszkoknak csak egy absztrakt megjelenése, azok kódja ugyanis nem itt található, hanem külön függvényekben, melyeket a virtuális gép taszkváltó funkciójával futtat az operációs rendszer. Itt csak az azokkal kapcsolatos paraméterek és a szükséges vezérlő funkciók kaptak helyet. A hagyományos, C nyelvű operációs rendszerekben ez az osztály a taszk irányító blokknak felelne meg (*Task Control Block, TCB*).

A *Clock* osztály a rendszer belső óráját valósítja meg. Tárolja a belső időt, amit egy függvényhívás hatására meg is tud adni. A `Tick()` metódus a minden óraütéskor elvégzendő funkciókat látja el. Ilyen például a késleltetések figyelése, új periódusok indításának figyelése. Ha szükséges, akkor a metódus újraütemezést kér a kerneltől. A `Tick()` metódust a virtuális gép óraütést kezelő interrupt függvénye hívja meg minden óra megszakításkor. Ezt az interrupt függvényt, valamint az óra felbontását az `Init()` metódus állítja be.

A *Forecast* osztály az aktuális terheltségi szintet állapítja meg. Egy struktúrában tárolja a legközelebb futó taszkok adatait, és ennek segítségével adja meg a következőleg futtatandó taszknak, hogy az melyik futási módot használja. A `FillTable()` a rendszer inicializálási fázisában feltölti a struktúrát adatokkal. Ezután minden taszk futtatásakor a `ShiftTable()` egy újabb taszk adatait veszi fel. A taszkok a `Decide()` hívással tudhatják meg, hogy melyik módban kell futniuk.

A *Log* osztály felelős a naplózásért. A taszkok és a kernel a `LogInsert()` hívással vehetnek fel egy eseményt a naplóba. Az osztály a naplót egy vektorban tárolja, amit egy bizonyos méret elérése után elküld a távoli monitorozó állomásnak, majd törli a tárolt információkat.

Az *Event* osztály egy eseményt valósít meg, ami tulajdonképpen egy változó tárolását jelenti. A taszkok események átadásával kommunikálhatnak egymással.

A *Communication* egy absztrakt osztály, tényleges példánya nem lesz. A kommunikációs objektumok ebből fognak öröklődni. Az általánosan használt műveletek prototípusait tartalmazza (post és pend).

A *Semaphore* osztály a legalapvetőbb kommunikációs objektumot valósítja meg, a szemafort. Az osztálynak egy változója van, ami a szemafor értékét mutatja. A `Pend()` hívással a taszk a szemafort használni szeretné. Ha a szemafor értéke pozitív, akkor az erőforrás használható, és a szemafor értéke eggyel csökken. Egyébként a hívás blokkol, és a taszknak meg kell várnia, amíg a szemafor felszabadul. A `Post()` jelzi a szemafor használatának a végét. Hatására a szemafor értéke eggyel nő.

A státusz információkat a *Status* osztály képviseli. Az osztály objektumai csak egy eseményt tudnak tárolni. Ez mindig az utoljára küldött adatot tartalmazza, azt

az olvasás nem törli. Adatot küldeni a `Post()` hívással lehet, a `Pend()` pedig az adatok olvasását végzi. Ha még nincs üzenet a Status típusú objektumban, akkor a taszk az olvasás hatására blokkolódik, egyébként nem.

A sor megvalósításáért a *Queue* osztály felelős. Az ilyen típusú objektumok több eseményt is tudnak tárolni (ezek számát a konstruktor paraméterében lehet majd megadni). A sorba írás akkor blokkol, ha az már tele van, egyébként az elküldött esemény a sor végére kerül. Az olvasás a sor elején levő eseményt adja vissza. Üres sor esetén blokkol.

Az itt felvázolt struktúrában egyik osztály sem tölt be nagyon erős központi szerepet. A rendszer inkább a komponensek együttműködéséből áll össze. A komponenseknek jól elkülönített, egyedi feladatuk van. Külön állnak a kommunikációs objektumok, amiket csak a taszkok megvalósításai fognak érdemben használni. Ezek inkább plusz szolgáltatásként lesznek jelen, nem fogják a rendszer szerves részét képezni.

Megoldható a rendszer elosztott működése is. Ehhez a különböző processzorokon működő taszkok kommunikációját kell lehetővé tenni. Ha minden elosztott részrendszerhez hozzárendelünk egy azonosítót, akkor meg lehet állapítani, hogy melyik kommunikációs objektum melyik rendszerben található. Ezután az írási és olvasási műveleteket úgy kell módosítani, hogy azok figyelembe vegyék, hogy távoli géppel akar-e a taszk kommunikálni. Ha a hívás paraméterében a helyi gép azonosítója szerepel, akkor minden a szokásos módon történik. Ha azonban egy távoli gépé, akkor a kérést a virtuális gép egy ilyen célú metódusával el kell küldeni a távoli gépnek, majd a választ továbbítani a hívást kezdeményező taszknak.

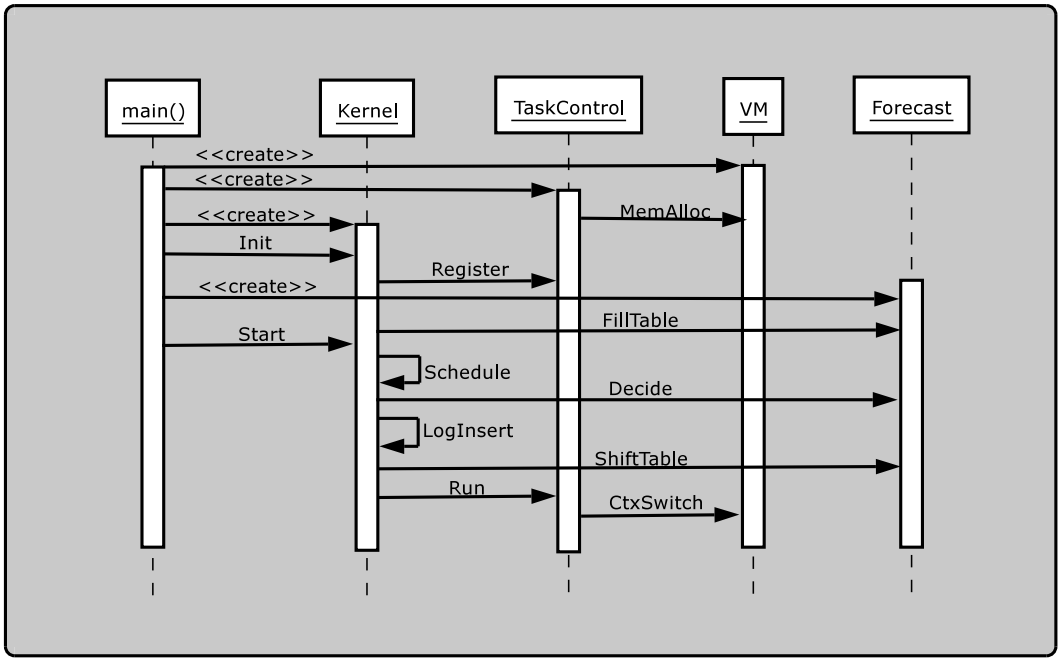
4.3.2. Dinamikus modellek

A valós idejű rendszerekben nagyon fontos szerepet játszik az időbeli viselkedés, ezért a dinamikus modellek többet mondhatnak a rendszer működéséről, mint statikus társaik. A tervezendő rendszer több komponensből áll; a teljes rendszer működése e komponensek együttműködéséből áll össze. Ez jellemző módon függvényhívásokban valósul meg. Ennek leghatékonyabb leírási módja a kommunikációs diagram. A következőkben a rendszer néhány fontosabb helyzetét ábrázoltam ilyen módon.

A 4.2. ábrán a rendszer indításának forgatókönyve látható.

A rendszer a `main()` függvénnyel indul, amely először a VM osztályt példányosítja. Ezután létrehozza a taszkokat leíró objektumokat, amelyek a virtuális gép segítségével memóriát foglalnak maguknak. Következő lépésként példányosítja a Kernel osztályt, majd meghívja annak `Init()` függvényét. Ez alapállapotba állítja a változókat, meghívja a taszkok regisztráló függvényét, valamint inicializálja az órát (az ábrán nincs feltűntetve). A `main()` ezután létrehoz egy Forecast típusú objektumot, ami a futási időket határozza majd meg. A `FillTable()` hívással a Forecast objektum feltölti adatokkal a benne tárolt struktúrát.

Ezek után a `Start()` hívás hatására elindul a rendszer működése. A Kernel meghívja az ütemezőt, ami kiválasztja a legközelebb futtatandó taszkot, majd az



4.2. ábra. Az operációs rendszer inicializálása

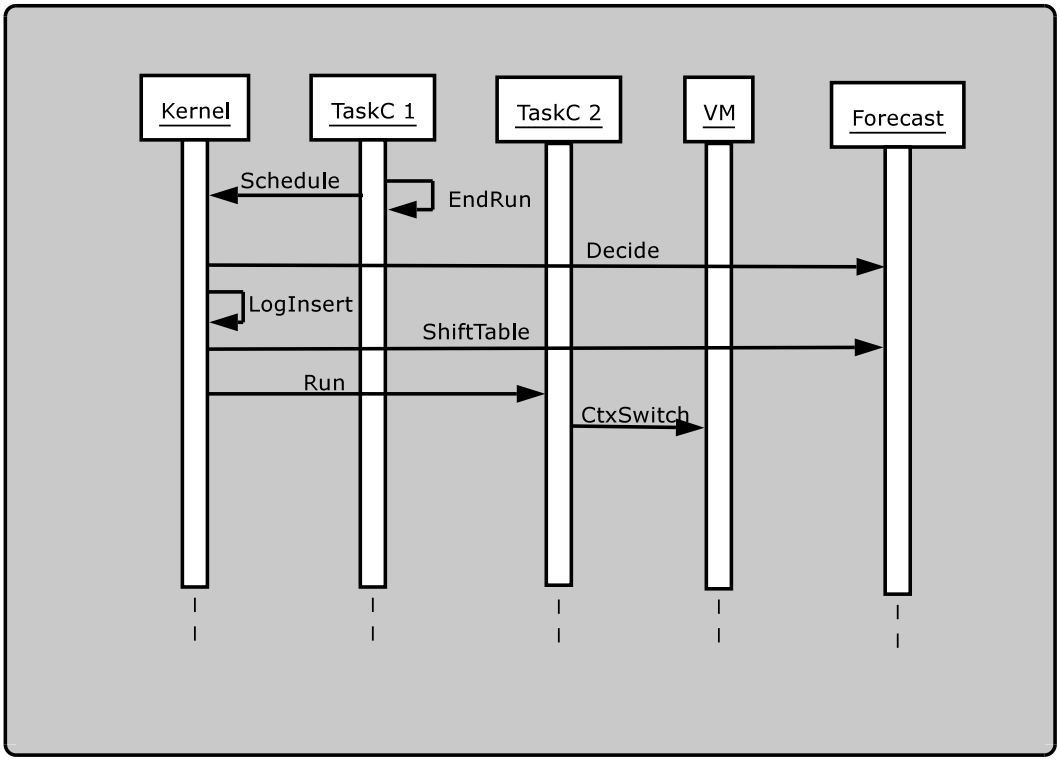
előrejelzőtől lekéri az engedélyezhető futási módot, utána pedig beleírja a naplóba az eseményeket (ez a hívás tulajdonképpen nem rekurzív, hanem a Log objektumnak szól). Az előrejelzési táblázatot frissíti, majd a futtatandó taszk `Run()` hívásával elkezdi annak végrehajtását. A futás a virtuális gép taszkváltó utasításának hatására fog ténylegesen elkezdődni.

A 4.3. ábra egy ütemezést mutat be. Újraütemezésre többféleképpen kerülhet sor:

- a futó taszk befejezi futását
- a futó taszk várakozni kényszerül
- új taszk lép be
- várakozó taszk futásra kész lesz

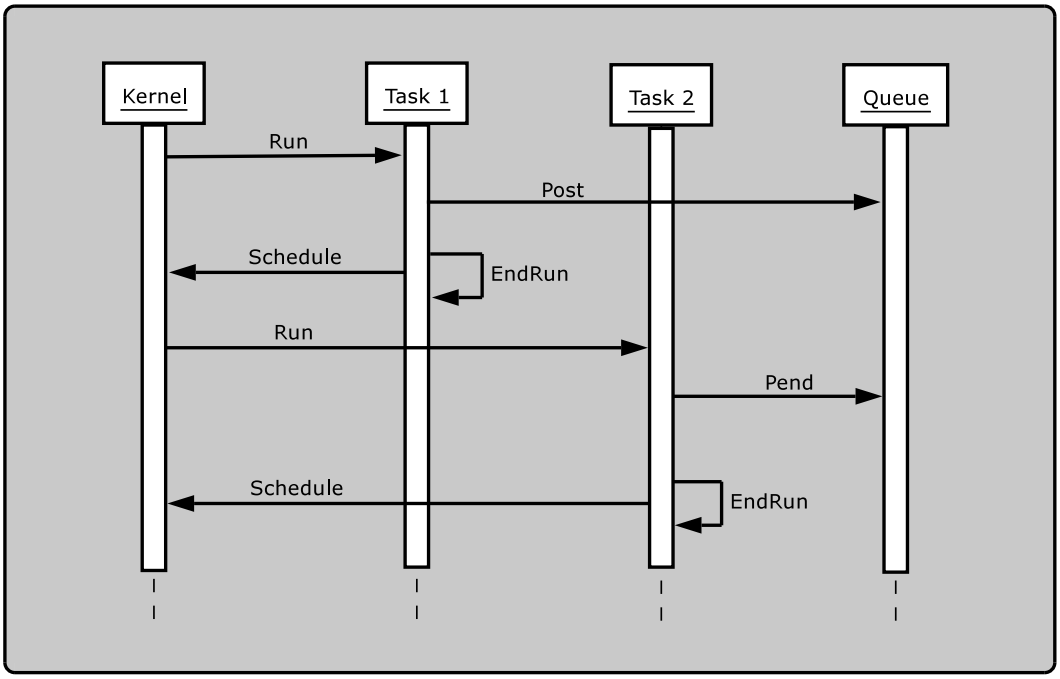
Az ábrán az az eset látható, amelynél az éppen futó taszk befejezi futását. Ekkor az `EndRun()` során a taszk meghívja a kernel ütemezőjét, ami eldönti, hogy melyik taszk fusson legközelebb, az előrejelző pedig azt, hogy milyen módban fusson. A kernel ezt követően meghívja az előrejelző frissítést végző metódusát, majd futtatja a kiválasztott taszkot a megkapott futási móddal. A taszkvezérlő egy taszkváltást kér, ami elindítja a futását.

A 4.4. ábra egy sor használatát mutatja be. Az ábrán először az egyes számú taszk fut, ami egy eseményt helyez el a sorban, majd befejezi futását. A kernel az előbb látott módon taszkot vált és a kettes számú taszkot kezdi futtatni, ami kiolvassa a sorba helyezett üzenetet, majd miután minden feladatát elvégezte, szintén befejezi a futást.



4.3. ábra. Egy ütemezés

Az operációs rendszer leállítása a `Stop()` hívással történik. Ennek hatására az operációs rendszer leállítja az éppen futó taszkot, felszabadítja a lefoglalt memóriát, visszaállítja az elállított megszakítás-vektorokat majd befejezi a futását.



4.4. ábra. Sor használata két taszk között

5. fejezet

A megvalósítás

5.1. A fejlesztői környezet kiválasztása

A fejlesztői környezet kiválasztásakor alapvetően DOS-os környezetben gondolkoztam, hiszen mind a Hartik, mind a μC/OS a DOS-ra épül. Ennek a megoldásnak az az előnye, hogy az operációs rendszer fejlesztésekor nem kell azzal törődni, hogy az hogyan induljon el, azaz nem kell külön rendszerindító részt írni. A rendszert DOS alól, a lefordított .exe fájl futtatásával indíthatjuk el. A megoldás hátránya az, hogy ez esetben csökken a hordozhatóság, hiszen a programnak szüksége lesz egy futó DOS-ra, ami már eleve x86 környezetet feltételez. A kódot azonban strukturálhatjuk úgy, hogy a hardver- és környezetfüggő részek külön egységbe kerüljenek, így egyszerűsítve az operációs rendszer portolását.

A célkörnyezet meghatározása után a használandó fordítót kellett kiválasztani. DOS alá az alábbi fordítók képesek fordítani:

- Borland C++ Builder
- Borland Turbo C++
- Microsoft Visual C++
- DJGPP

A fenti fordítók közül a Borland Turbo C++ és a DJGPP szabadon elérhető, így e kettő közül választottam ki a használandó eszközt.

A Turbo C++ régebbi fejlesztés, 1992-ben készítették. Valós módú kódot generál, ami a régebbi (386 előtti) processzorokon is működőképes. Nem tartalmazza a szabványos sablon könyvtárat (STL). A fejlesztői környezete jól használható.

A DJGPP [DJGPP] a GNU C fordítóinak (GCC, G++) DOS-os portja. A fejlesztése még most is tart. A rendszer nyílt forráskódú, így amellett, hogy ingyenesen elérhető, még a forrása is szabadon letölthető. Az ezzel fordított programok védett módúak lesznek. Tartalmazza a szabványos sablon könyvtárat. Van saját fejlesztői környezete, az Rhide, amely nagyban hasonlít a Turbo C++-éra. Nagy előnye, hogy

a GNU fordítóit használja, mert ezek rengeteg processzorra képesek fordítani, és elérhetőek az összes Linux és UNIX disztribúcióban. Vagyis az ezzel generált program a hardverfüggő részt leszámítva hordozható lesz.

A fenti tulajdonságok alapján a DJGPP mellett döntöttem. A fejlesztést Windows XP alatt, DOS ablakban végeztem, leginkább annak többfeladatos képességei miatt. A fejlesztéshez a DJGPP 2.03-as, a GCC 3.3.3-as, a G++ 3.3.3-as és a GNU Assembler 2.14-es verzióját használtam.

5.2. A végleges felépítés

A rendszertervben felvázoltam a program előzetes tervét. Ez a tényleges megoldás során, ha nem is óriási mértékben, de változott. Az alábbiakban először az operációs rendszer szerkezetét fogom ismertetni, majd az egyes funkciók működésére térek ki részletesen.

Az operációs rendszer felépítését az 5.1. ábra mutatja. Az ábrán látszik, hogy az alapvető osztályok, és azok kapcsolatai nem változtak nagy mértékben. Különbségeket leginkább a metódusokban lehet találni. Egyes osztályok újabbakkal egészültek ki, valamint egy metódust át kellett helyezni egy másik osztályba (`RegisterTask()`).

Két új osztályt is be kellett vezetni, melyek a távoli monitorozó állomással [Bartha04] tartják a kapcsolatot. Az egyik az Actuator, ami beolvassa a távoli számítógéptől kapott utasításokat, majd végrehajtja. A másik a Monitor, ami a Log osztály egy csomagoló osztálya. A monitorozó egység számára érthető formában írja a bejegyzéseket a naplóba.

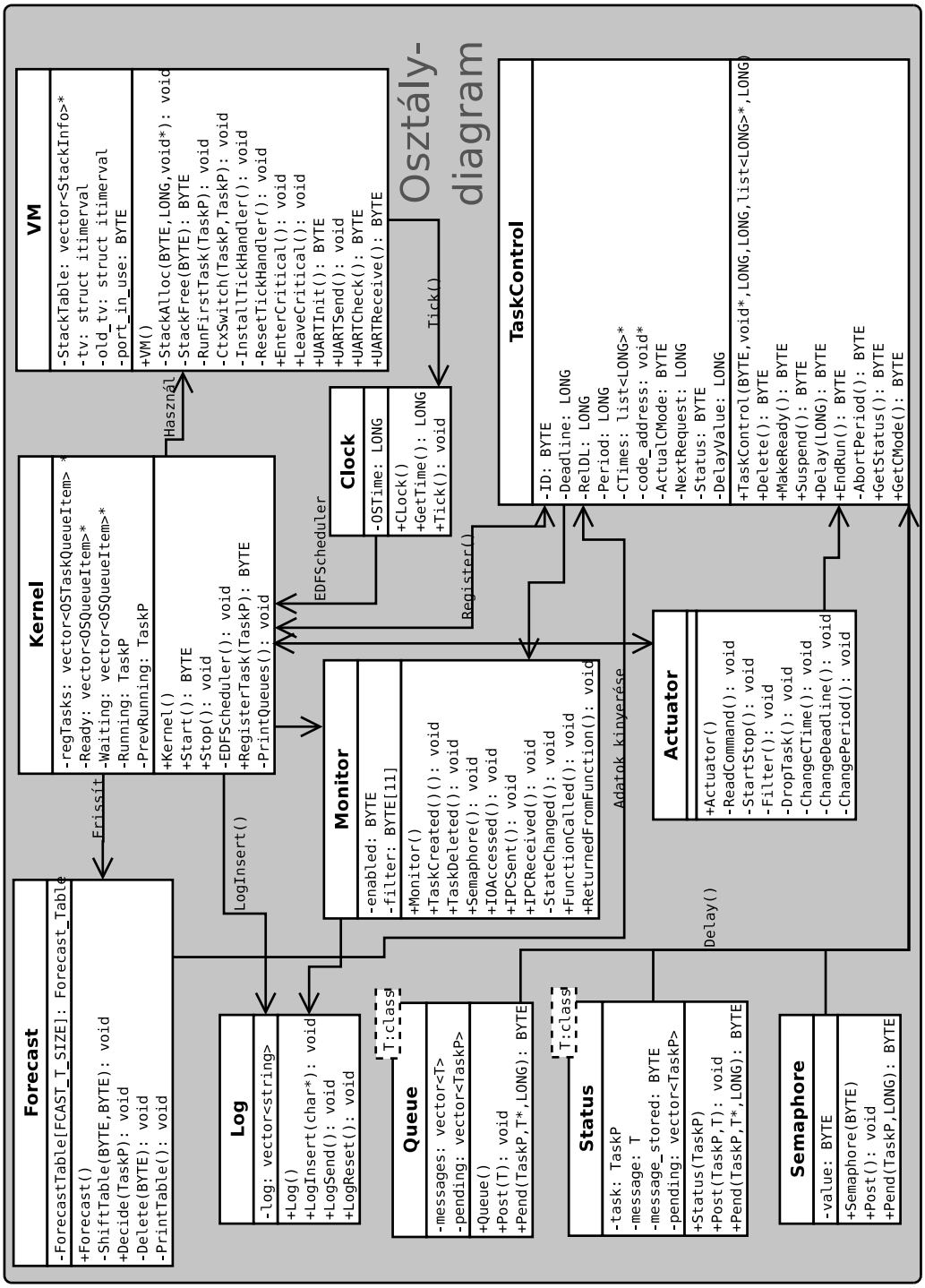
5.2.1. A Kernel osztály

Ez az osztály, mint azt az előző fejezetben említettem az alapvető operációs rendszer funkciókért felelős. Ilyen például az operációs rendszer indítása és leállítása, az ütemezés, valamint a taszkok nyilvántartásba vétele. Először az osztály attribútumait, majd sorban a metódusait mutatom be.

Attribútumok

A Kernel osztály adatszerkezetei nagyrészt a regisztrált taszkokkal kapcsolatosak.

regTasks OSTaskQueueItem típusú vektor. A mérete adott, tehát a regisztrálható taszkok száma fordítási időben dől el. Ezt a `config.h` konfigurációs fájl `MAX_TASK_NO` konstansával lehet beállítani. A vektor egyes elemei az adott taszk azonosítóját tartalmazzák, valamint egy mutatót a taszk TaskControl objektumára.



5.1. ábra. A végleges osztály diagram

ready OSQueueItem típusú vektor. A mérete szintén meghatározott, ugyanaz a konstans állítja, mint a regTasks vektorét. Itt tárolja az objektum a futásra kész taszkokat. A vektor egy eleme tartalmazza a taszk azonosítóját, határidejét, valamint egy mutatót a taszk TaskControl objektumára.

waiting Felépítése ugyanaz, mint a ready attribútumé. Az objektum itt tárolja a várakozó taszkokat.

Running Mutató egy TaskControl objektumra. Ez mindig az éppen futó taszkra mutat.

PrevRunning Szintén egy TaskControl objektumra mutató mutató. Ez az előzőleg futott taszkot határozza meg.

Az attribútumokat az 5.1. táblázat foglalja össze.

5.1. táblázat. A Kernel osztály attribútumai

Attribútum	*Típus*	*Tagváltozók*	
regTasks	vector<OSTaskQueueItem>	TaskP BYTE	Task ID
ready	vector<OSQueueItem>	TaskP BYTE LONG	Task ID Deadline
waiting	vector<OSQueueItem>	TaskP BYTE LONG	Task ID Deadline
Running	TaskP	–	
PrevRunning	TaskP	–	

Kernel()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Ez az osztály konstruktora. Először az operációs rendszer sorainak foglal memóriát. Mindegyik akkora méretű lesz, mint ami a `MAX_TASK_NO` konstansban szerepel. Ezután létrehozza az üresjárati taszkot és beregisztrálja azt a `RegisterTask()` metódussal. Az üresjárati taszk azonosítója 255, számítási ideje nulla, nem periodikus, határideje szintén nulla, csakúgy, mint a késleltetése.

RegisterTask(TaskP)

- *Bemeneti paramétere*: a kívánt taszk vezérlő objektumára mutató mutató.
- *Visszatérési értéke*: `NO_MORE_TASK_SPACE`, `ID_0_NOT_ALLOWED`, `NO_ERROR`.

A metódus a paraméterben jelzett taszkot regisztrálja be a kernel objektumba. Először megvizsgálja, hogy van-e még hely a sorokban a taszk számára. Ha nincs, akkor `NO_MORE_TASK_SPACE` értékkel tér vissza. A metódus a nulla azonosítóval rendelkező taszkokat sem engedélyezi, mivel ennek különleges szerepe lesz a Forecast osztálynál. Nulla értékű azonosító esetén az `ID_0_NOT_ALLOWED` értékkel tér vissza. Ha egyik helyzet sem áll fent, akkor először beilleszti a taszkot (létrehozva a megfelelő struktúrát) a *regTasks* vektor elejére, majd törli a vektor utolsó elemét, hogy a mérete ne változzon. Ezt követően, attól függően, hogy a taszk Status attribútuma `READY` vagy `WAITING`, elhelyezi vagy a futásra kész, vagy a várakozó listában. A futásra kész listába a taszkok a határidejük alapján csökkenő sorrendben kerülnek be. A lista elején mindig a legközelebbi határidejű taszk található. Az EDF ütemezőnek így nem kell végigkeresni a listát, hogy ezt megtalálja. Ha a taszk regisztrációja az operációs rendszer indítása után történt, akkor meghívja az ütemezőt. Végül a metódus `NO_ERROR` értékkel tér vissza.

Start()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: BYTE típusú.

Az operációs rendszer működést indító metódus. Először feltölti az előrejelző objektumot a szükséges adatokkal (meghívja a forecast objektum `ShiftTable` metódusát a megfelelő paraméterekkel). Ezután meghívja az ütemezőt, ami kiválasztja a futtatandó taszkot. Ezt követi a hardver óra megszakítását kezelő rutin beregisztrálása, amit a vm objektum `InstallTickHandler` metódusának meghívásával ér el. Végül futtatja az első futásra kész taszkot (ezt az EDF ütemező választotta ki, és a *Running* attribútum tartalmazza).

Stop()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az operációs rendszer leállítására szolgál. Visszaállítja a hardver óra megszakítás kezelő rutinját az eredetire a vm objektum `ResetTickHandler` metódusának meghívásával. Ezután kilép a programból egy `exit(-1)` hívással.

EDFScehduler()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az ütemező, ami a futtatandó taszkot választja ki. A metódus neve is mutatja, hogy EDF (Earliest Deadline First) ütemezőről van szó, azaz mindig a legközelebbi határidejű taszk fut.

A metódus először beállítja a *PrevRunning* mutató értékét az eddig futtatott taszkra. Ezután a ready vektor első eleme lesz a futtatandó taszk (mivel a vektor mindig határidő szerinti sorrendben tárolja a taszkokat), erre állítja át a *Running* attribútum értékét. Ezután a forecast objektum `Decide` hívásával meghatározza, hogy a taszk milyen futási módot használjon (kivéve, a taszk állapota `RUNNING`, vagy az üresjárati taszkról van szó). Ha a taszkot eldobja az előrejelző objektum, akkor újat keres. Az így kiválasztott taszk állapotát `RUNNING`-ra állítja, majd ha nem az első taszk futtatásánál tartunk (a `FirstSchedule` attribútum nem nulla), akkor meghívja a kontextusváltó függvényt (`vm.CtxSwitch()`). Ezzel az új taszk elkezd futni.

PrintQueues()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Hibakeresést elősegítő függvény, ami csak akkor fordítódik le, ha a `config.h` fájlban definiáltuk a `DEBUG` konstanst. A metódus végigmegy a kernel objektum vektorain, és kiírja a szabványos kimenetre azok tartalmát.

5.2.2. A TaskControl osztály

Ez az osztály tartalmazza a taszkok paramétereit, valamint a összegyűjti a taszkokkal elvégezhető műveleteket. Egy nem objektum orientált operációs rendszerben a taszk vezérlő blokknak (*TCB, Task Control Block*) felelne meg. A taszk tényleges kódját nem tartalmazza, csak egy hivatkozást rá.

Attribútumok

ID A taszk azonosítóját tartalmazza. BYTE (1 bájt) típusú, ami maximum 256 különböző taszkot feltételez; ebből kettő foglalt (a 255 és a 0).

Deadline A taszk határideje. Ez az abszolút értékben vett határidő. A taszknak be kell fejeznie a futását mire az operációs rendszer órája eléri ezt az értéket. LONG (4 bájt) típusú.

RelDL A taszk relatív határideje. Szintén LONG típusú. Az abszolút határidőt úgy kaphatjuk meg, hogy a taszk futási kérelméhez hozzáadjuk az értékét.

Period A taszk periódusideje. A taszk ennyi idő elteltével fog újra futásra jelentkezni. Ha az értéke nulla, akkor aperiodikus taszkról van szó. A típusa LONG.

CTimes A taszk számítási ideit tartalmazó lista. Típusa egy LONG elemekből álló listára mutató (`list<LONG>*`). A lista elemei a taszk különböző futási módjaihoz tartozó számítási időket tartalmazzák. Egy taszk 256 féle számítási móddal rendelkezhet (mivel az aktuális módot kijelölő változó BYTE típusú).

ActualCMode Az éppen használt számítási módot tartalmazó attribútum. A CTimes lista egy elemét jelöli. A nulla érték az első számítási módot jelenti, az egy a másodikat, és így tovább. BYTE típusa van, tehát legfeljebb 256 számítási módot tud kezelni.

NextRequest A taszk következő futási kérelme. Az operációs rendszer ekkor fogja legközelebb futtatni a taszkot. Értéke az operációs rendszer indulásakor a késletetés lesz (offset). Ezután periodikus taszk esetén mindig a legutolsó futási kérelem és a periódus összege. LONG típusú.

Status A taszk állapotát jelző változó. Egy taszk `READY`, azaz futásra kész, `WAITING` azaz várakozó, `RUNNING` azaz futó, vagy `ZOMBIE`, azaz törlendő állapotú lehet. BYTE típusú.

DelayValue A taszk késleltetése. Egy taszk késleltetésénél ez a változó tárolja a késleltetés mennyiségét. Minden óraütésnél csökken egyel az értéke. Amíg nem nulla, a taszk várakozó állapotban kénytelen maradni. Ha nullára csökken, az operációs rendszer újra futásra kész állapotba helyezi a taszkot.

Az osztály attribútumait az 5.2. táblázat foglalja össze.

5.2. táblázat. A TaskControl osztály attribútumai

Attribútum	*Típus*
ID	BYTE
Deadline	LONG
RelDL	LONG
Period	LONG
CTimes	list<LONG>
ActualCMode	BYTE
NextRequest	LONG
Status	BYTE
DelayValue	LONG

TaskControl(BYTE, void*, LONG, LONG, list<LONG>, LONG)

- *Bemeneti paraméterei*: azonosító, a taszk kódjának címe, határidő, periódusidő, számítási idők, ofszet.
- *Visszatérési értéke*: nincs.

A TaskControl osztály konstruktora. A bemeneti paraméterek alapján feltölti az osztály attribútumait a megfelelő értékekkel. A határidő értéket a *RelDL* attribútum kapja meg. A *NextRequest* értéke az ofszettel lesz egyenlő. A késleltetés nulla lesz. Az éppen használt számítási mód az első, azaz a legrészletesebb lesz. A taszk státusza az ofszet értékétől függ. Ha az megegyezik az éppen aktuális operációs rendszer idővel, akkor a taszk futásra kész, azaz `READY` lesz. Ekkor az abszolút határidő is megkapja a megfelelő értékét. Ellenkező esetben a taszk várakozó állapotba kerül, azaz `WAITING` lesz.

Delete()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `NO_SUCH_TASK`, `NO_ERROR`.

Az adott objektumhoz tartozó taszkot törli az operációs rendszer nyilvántartásából. Először megkeresi a taszkot a regisztrált taszkokat tartalmazó listában, majd törli innen. Ezután először a futásra kész listát nézi végig. Ha itt megtalálja a taszkot, akkor törli, és egy új elemet fűz a listához. Ha nem találja, akkor a várakozó listát nézi végig, és innen törli. Ha a törlendő taszk éppen fut, akkor az ütemezőt is meghívja, hogy az új taszkot ütemezzen. A taszk állapotát `ZOMBIE`-ra állítja. A memóriát a rendszer a következő óraütéskor szabadítja fel. Ha a taszk nincs a nyilvántartásban, akkor a metódus `NO_SUCH_TASK`, egyébként pedig `NO_ERROR` üzenettel tér vissza.

MakeReady()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `NOTHING_TO_DO`, `NO_ERROR`.

Az adott taszkot futásra kész állapotba helyezi. Ehhez először megkeresi a várakozó listában. Ezután törli innen, és áthelyezi a készenléti listába, mégpedig a határidejének megfelelő helyre. A taszk státuszát átállítja futásra készenre, valamint az esetleges késleltetést is megszűnteti. Ha a taszk nincs a várakozó listában, akkor a metódus `NOTHING_TO_DO`, egyébként pedig `NO_ERROR` üzenettel tér vissza.

Suspend()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `NOTHING_TO_DO`, `NO_ERROR`.

Felfüggeszti az adott objektumhoz tartozó taszkot. Ehhez először megkeresi a készenléti listában. Ha megtalálta, akkor törli innen, és áthelyezi a várakozó listába, a taszk státuszát pedig átállítja várakozóra. Ha a taszk nem volt a futásra kész listában, akkor a metódus `NOTHING_TO_DO`, egyébként pedig `NO_ERROR` üzenettel tér vissza.

Delay(LONG)

- *Bemeneti paramétere*: a késleltetés ideje.
- *Visszatérési értéke*: `NOTHING_TO_DO`, `NO_ERROR`.

A taszkot a paraméterben megadott idővel késlelteti. Tulajdonképpen ugyanazt csinálja, mint a `Suspend()`, annyi különbséggel, hogy itt a taszk késleltetés attribútumát is beállítja a megadott értékre. Ha a taszkot nem találja meg, akkor a metódus `NOTHING_TO_DO`, egyébként pedig `NO_ERROR` üzenettel tér vissza.

EndRun()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `TASK_NOT_RUNNING`, `NO_ERROR`.

Az éppen futó taszknak kell meghívnia a futás végén. Ezzel jelzi, hogy elvégezte a feladatát. A metódus először ellenőrzi, hogy tényleg az adott taszk fut-e. Ha nem, akkor `TASK_NOT_RUNNING` üzenettel tér vissza. Ha a taszk az éppen futó, akkor az eljárás felfüggeszti, majd meghívja az ütemezőt. Aperiodikus taszk esetén többet nem fog futni a taszk. Periodikus esetben viszont az operációs rendszer akkor fogja indítani a következő periódust, amikor a belső óra értéke eléri a *NextRequest* attribútum értékét (ezt a taszk futása előtt állítja be az operációs rendszer). Végül a metódus `NO_ERROR` üzenettel tér vissza.

AbortPeriod()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `NOTHING_TO_DO`, `NO_ERROR`.

Az adott objektumhoz tartozó taszk aktuális periódusát szakítja félbe. Ez abból áll, hogy leveszi a készenléti listáról a taszkot, és áthelyezi a várakozó listára. A taszk státuszát `WAITING`-re állítja. A taszk következő futtatási helyét a periódus elejére állítja vissza. Majd ha a taszk éppen futott, meghívja az ütemezőt. A metódus csak futásra kész taszkoknál használható. Ha a taszk nem volt a futásra kész listában, akkor a metódus `NOTHING_TO_DO`, egyébként pedig `NO_ERROR` üzenettel tér vissza.

GetStatus()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `READY`, `WAITING`.

Az objektumhoz tartozó taszk státuszát adja vissza (a *Status* attribútum értékét). Ez lehet `READY`, vagy `WAITING`.

GetCMode()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: BYTE típusú.

A taszkok hívják a futás kezdetén. Visszatérési értékében megadja az *ActualCMode* attribútum értékét, vagyis azt, hogy a taszk melyik futási módot használhatja. A nulla érték tartozik az első módhoz, az egy a másodikhoz, és így tovább.

5.2.3. A Clock osztály

Az időt kezelő osztály. Feladata a rendszeridő adminisztrálása és a minden óraütésben elvégzendő feladatok ellátása.

Attribútumok

OSTime Az operációs rendszer belső idejét tároló változó. Típusa LONG.

Az osztály attribútumait az 5.3. táblázat foglalja össze.

5.3. táblázat. A Clock osztály attribútumai

Attribútum	*Típus*
OSTime	LONG

Clock()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az osztály konstruktora. Az egyetlen funkciója az, hogy az *OSTime* attribútumot lenullázza.

GetTime()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: LONG típusú.

Visszatérési értékében megadja a rendszeridőt (azaz az *OSTime* attribútum értékét).

Tick()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

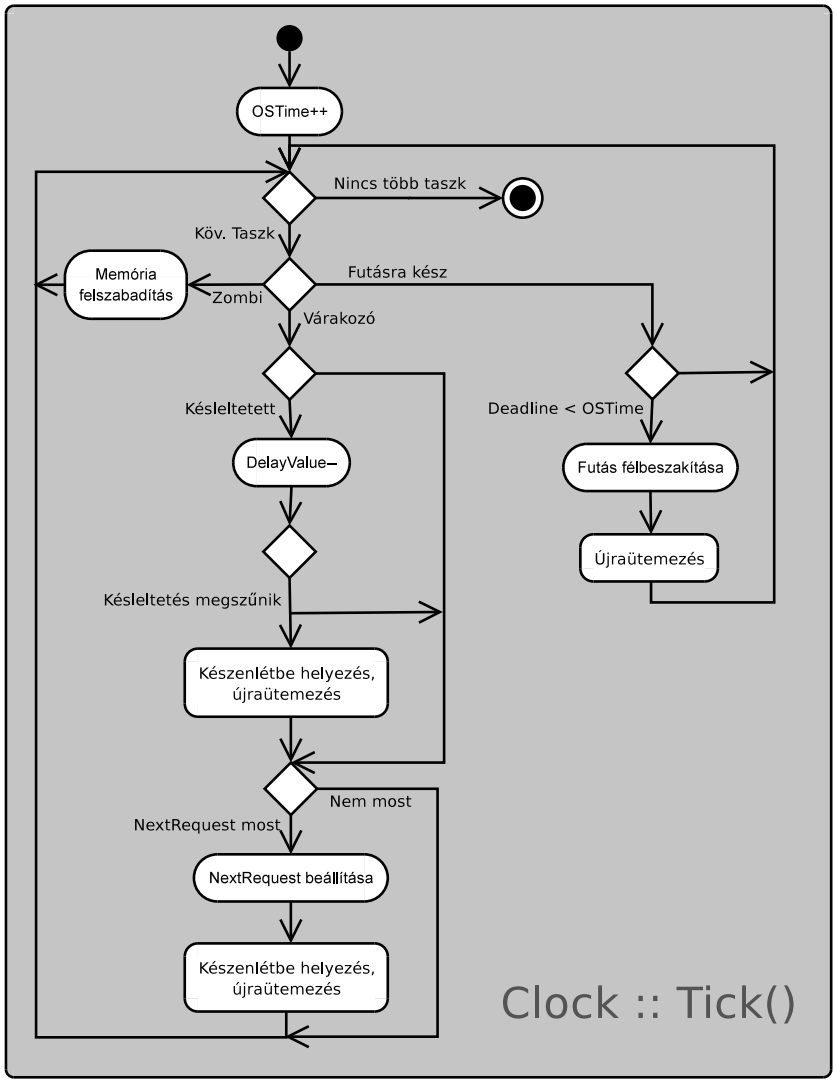
A metódus az óraütésenként elvégzendő feladatokat látja el. Elsőként növeli a rendszeridőt eggyel. Ezután végigmegy az összes regisztrált taszkon (kihagyva az üresjárati taszkot).

Ha a taszk várakozó állapotban van, akkor a metódus először ellenőrzi, hogy késleltetett taszkról van-e szó. Ha a taszk *DelayValue* attribútuma nem nulla, akkor azt eggyel csökkenti. Ha ennek hatására nulla lesz, akkor az eljárás futásra kész állapotba helyezi a taszkot, valamint egy állapotváltozóban eltárolja, hogy a futás végén meg kell hívnia az ütemezőt. Ezután azt ellenőrzi, hogy a taszk *NextRequest* értéke megegyezik-e a rendszeridővel. Amennyiben igen, ugyanúgy készenléti állapotba helyezi a taszkot, és a futás végén meg fogja hívni az ütemezőt.

Ha a taszk futásra kész, akkor azt kell ellenőriznie, hogy a taszk a futása során túllépte-e a határidejét. Amennyiben igen, félbeszakítja a taszkot, és meghívja az ütemezőt.

Ha a taszk törlendő (azaz `ZOMBIE` állapotú), akkor először felszabadítja a vermét, majd a taszkleíró objektumot.

A metódus működését az 5.2. ábra illusztrálja.



5.2. ábra. A Tick() metódus működése

5.2.4. A Forecast osztály

Ez az osztály felelős a taszkok számítási módjainak kezeléséért. Az osztály egy táblázatban tárolja a legközelebb futó taszkokat (ezek száma paraméterben megadható), és ezen adatok alapján dönt arról, hogy melyik taszk milyen módban fusson.

Attribútumok

ForecastTable[FCAST_T_SIZE] Ez a mátrix tárolja a legközelebb futó taszkok adatait. A méretét a `config.h` fájlban található `FCAST_T_SIZE` konstans beállításával lehet megváltoztatni. A típusa Forecast_Table. Egy eleme tartalmazza

- a taszk azonosítóját
- egy mutatót a taszkra
- a használt számítási mód sorszámát
- egy mutatót a kiválasztott számítási mód futási idejére
- azon taszkok számítási idejének összegét, melyek a taszk futásába beleszólhatnak (beleértve a taszkot magát is).
- a taszk határidejét
- és a taszk indulási idejét.

Az osztály attribútumait az 5.4. táblázat foglalja össze.

5.4. táblázat. A Forecast osztály attribútumai

Attribútum	*Típus*	*Tagváltozók*	
ForecastTable[]	Forecast_Table	BYTE	ID
		vector<OSTaskQueueItem>::iterator	Task
		BYTE	CModeNumber
		list<LONG>::iterator	CMode
		LONG	SumC
		LONG	Deadline
		LONG	StartTime

Forecast()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az osztály konstruktora. Tényleges feladata nincsen.

ShiftTable(BYTE, BYTE)

- *Bemeneti paramétere*: eltolás kezdőpozíciója, eltolás mennyisége.
- *Visszatérési értéke*: nincs.

A metódus feladata az, hogy az előrejelzési táblázatot frissítse a megadott paraméterek alapján. Az első paraméterben megadott pozíciótól kezdve a táblázatot eltolja a második paraméterben kapott értékkel. Amelyik oszlopoknál az eltolás már túlmutatna a táblázaton, azokat új adatokkal tölti fel.

Az operációs rendszer indulásakor a kernel objektum ezzel a hívással tölti fel a táblázatot. Paraméterként ekkor nullát, illetve ötöt ad át. Vagyis a táblázat összes oszlopa új értéket kap. Amikor egy taszk elkezdi a futását, kikerül a táblázatból, és a táblázatot eggyel arrébb kell tolni. Ez egy új elem felvételét jelenti. Ekkor a

paraméterezés a következő: pozíciónak az algoritmus az elindult taszk táblázatbeli pozícióját kapja, mennyiségként pedig egyet.

Az algoritmus tehát egy for ciklussal végigmegy a kapott pozíciótól kezdve a táblázat oszlopain. Amíg az eltolás a táblázat egyik oszlopára mutat, addig a soron következő oszlop egyszerűen megkapja az arrébb levő oszlop értékét. Ha azonban a pozíció és az eltolás összege meghaladja az táblázat méretét, akkor új értéket kell felvenni.

Ehhez először meg kell határozni, hogy melyik taszk fog a legközelebb indulni (az utolsó táblázatban szereplő időponthoz képest). Aperiodikus taszkok esetében ezt egyszerűen megkaphatjuk a taszk attribútumából. Periodikus taszk esetén azonban meg kell határozni, hogy mikor kezdődik a taszk azon periódusa, amelyik az utolsó indulási időhöz a legközelebb esik, majd ezzel kell számolni. Az algoritmus az alábbi képlet alapján számol:

$$\text{Periódus kezdet} = \text{Indulási idő} + \left(\frac{\text{Legnagyobb indulási idő a táblázatban} - \text{Indulási idő}}{\text{Periódus}} + 1\right) \text{Periódus}. \quad (5.1)$$

Az algoritmus tehát megnézi, hogy a periodikus taszk hányadik periódusa esik a táblázat legnagyobb indulási ideje elé, majd az ezt követő periódus kezdetét veszi. Így az algoritmus minden taszkhoz meg tudja határozni az indulási idejét, vagyis ki tudja választani ezek közül a legkisebbet.

Ha nincsen taszk, ami futhatna (az algoritmus az üresjárati taszkot kihagyja a számításból), akkor az algoritmus a táblázatot ezt jelző adatokkal tölti fel. Indulási időnek az aktuális rendszeridőt írja be, azonosítónak pedig a legnagyobb azonosítónál egyel nagyobbat. A számítási mód `DROPPED_TASK` lesz, azaz az eldobást jelző érték. A többi adat nulla lesz.

A táblázat adatai, a futási móddal kapcsolatos mezőket kivéve, egyértelműen kitölthetőek a taszkvezérlő objektumok adatai alapján. A taszkra mutató iterátort pedig már meghatározta a metódus, amikor a taszkot választotta ki legközelebb futónak. Így ezeket az adatokat egyszerűen ki lehet tölteni.

Egyszerű esetnek számít, amikor a táblázat első oszlopát tölti ki az algoritmus, hiszen ekkor nincs másik taszk a táblázatban, amelyre figyelemmel kéne lenni. Így tehát a taszk használhatja az első futási módot. A számítási idők összege pedig csak a taszk számítási ideje lesz.

Ha azonban későbbi oszlop kerül sorra, akkor a taszk futási módját is meg kell határozni. Ehhez először ki kell számítani azon taszkok számítási ideinek összegét, amelyek hatással lehetnek a taszk futási idejére. Az algoritmus jelenleg úgy működik, hogy a táblázatban előbb szereplő taszkok közül az (5.2) feltételt teljesítő taszkok számítási idejét adja hozzá az összeghez (azokkal a futási időkkel, melyeket az algoritmus korábban már meghatározott).

$$\text{Mostani taszk határideje} < \text{Korábbi taszk határideje} \leq \text{Mostani taszk indulási ideje} \quad (5.2)$$

Az így kiszámított összeghez hozzáadja a taszk legnagyobb számítási idejét, és megnézi, hogy az belefér-e az indulás és a határidő közti időbe. Ha nem, akkor az eggyel alacsonyabb szintű futási módot választja, és újra megnézi, hogy az belefér-e

a keretbe. Ezt egészen addig folytatja, amíg egy megoldást nem talál, vagy el nem fogynak a futási módok.

Ha nem talált megoldást, akkor az algoritmus a sorrendben hátrébb levő taszkok számítási idejét próbálja meg csökkenteni. Értelemszerűen csak azokét, melyek számítási ideje szerepel az előbb kiszámított összegben. Az algoritmus egyesével végigmegy az összes ilyen taszkon, és mindegyiknél végigpróbálja a futási módokat csökkenő sorrendben. Egyszerre egy taszk futási módján változtat (a meghatározandó taszk ekkor már biztosan a legkisebb módban fog futni).

Ha talál egy megoldást, akkor az korábbi taszk értékeit módosítja a táblázatban. Ezután az összes olyan oszlopban módosítja a futási idők összegét, ahol ez a taszk szerepelt (lásd az (5.2) feltételt). A most meghatározandó taszk pedig a legrövidebb futási módot fogja kapni.

Ha ilyen módon sem volt megoldás, akkor a taszkot el kell dobni. Ezt a *CModeNumber* mező `TASK_DROPPED` értéke jelzi. Hibakereső módban (lásd `config.h`) az algoritmus a futás végén kiírja a táblázatot a szabványos kimentre. Az algoritmus vázlatos működését az 5.3. ábra mutatja be.

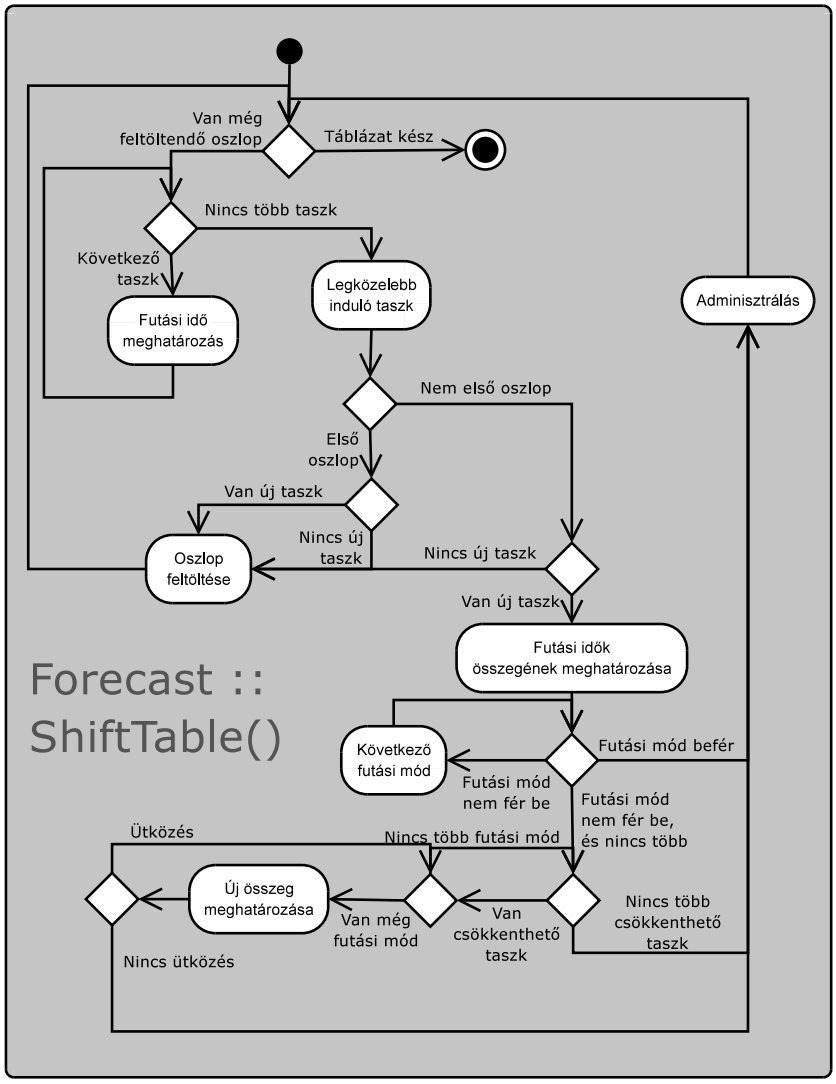
Decide(TaskP)

- *Bemeneti paramétere*: taszkvezérlő mutató.
- *Visszatérési értéke*: nincs.

Annak eldöntésére szolgál, hogy egy adott taszk milyen futási módban fusson. Az ütemező hívja meg a metódust, amikor egy új taszkot ütemez. Az algoritmus a döntés eredményét a paraméterként kapott taszkvezérlő *ActualCMode* attribútumában helyezi el. Ezt a taszkok a kernel objektum `GetCMode()` hívásával kaphatják meg.

Az algoritmus csak futásra kész állapotú taszkok esetében fut le, egyébként nem csinál semmit. Először végignézi az előrejelző táblázatot, hogy a kapott taszk szerepel-e benne. Egy taszk akkor szerepel a táblázatban, ha az azonosítója megegyezik a táblázatban szereplőével, valamint a határideje kisebb az operációs rendszer idejénél. Ezt azért szükséges kikötni, mert előfordulhat, hogy egy taszk több, későbbi periódusával is szerepel a táblázatban. Ekkor az algoritmusnak csak a megfelelő periódust szabad megtalálnia. Ha megvan a taszk, akkor a taszkvezérlő *ActualCMode* attribútuma megkapja a táblázatban szereplő *CModeNumber* értéket. Ha a taszk nem a táblázat első helyén szerepelt, akkor a metódus a számítási idejét hozzáadja az előtte szereplő taszkok számítási idő összegeihez, mivel a taszk előttük indult náluk, de a számítási ideje nem szerepel az összegekben (mert a táblázatban később helyezkedett el).

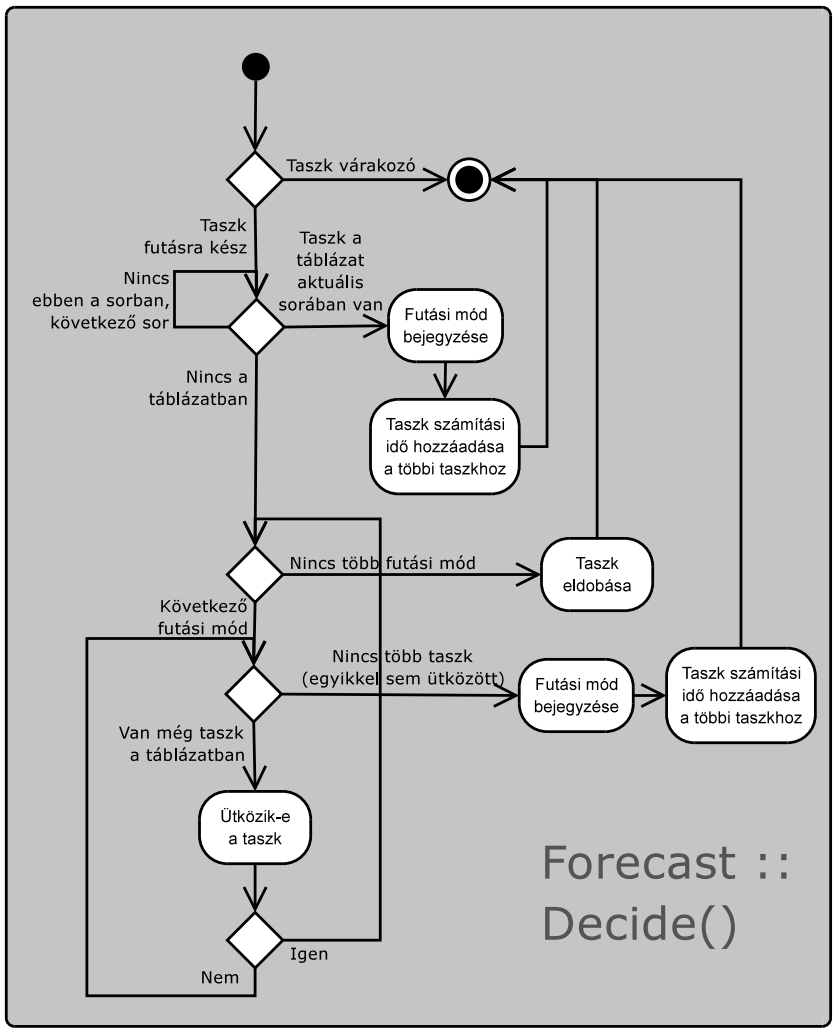
Ha a taszk nem található meg a táblázatban, akkor a metódusnak el kell döntenie, hogy milyen futási módot használjon. Ehhez csökkenő sorrendben végignézi a taszk számítási módjait. A számítási időt a táblázat minden olyan taszkjának *SumC* értékéhez hozzáadja, melyeknek határideje kisebb a taszk határidejénél. Ha van



5.3. ábra. A ShiftTable() metódus működése

olyan taszk, amelynél az így kapott számítási összeg már nem fér bele a határidő és az indulás közti időbe, akkor az adott számítási mód nem használható. Ha egyik számítási móddal sem jön létre ütközésmentes helyzet, akkor a taszkot el kell dobni. Ha viszont van egy megfelelő futási mód, akkor a taszk azt fogja használni. Ekkor a metódus hozzáadja a számítási idejét a táblázat minden olyan taszkjához, melynek indulási ideje korábban van, mint a most vizsgált taszk határideje (az ugyanis nem szerepel a táblázatban).

A metódus működését az 5.4. ábra illusztrálja.



5.4. ábra. A Decide() metódus működése

5.2.5. A VM osztály

Az osztály a hardverközeli kódokat csoportosítja, ezzel könnyítve meg az operációs rendszer portolását más platforomokra. Külön fájlok (`vm_x86.cc` és `vm_x86.h`) tartalmazzák az osztályt. Az fájlok az osztály mellett több hardverfüggő konstanst is tartalmaznak, leginkább a soros port kezelésével kapcsolatban. Itt definiáltam az operációs rendszerben használt változó típusokat is, mert a C nyelv változói különböző rendszerekben különböző értékeket vehetnek fel. A `BYTE` egy bájtos adatot, a `WORD` két bájtos adatot, míg a `LONG` négy bájtos adatot jelöl.

Attribútumok

StackTable vector<StackInfo> típusú. A taszkok vermeit tároló lista. Tárolja a taszkok azonosítóját, a verem tetejét, a verem kezdőcímét, méretét, a taszkok

kezdőutasításainak címét, valamint a taszkok állapotát tároló tömböt.

tv Típusa stuct itimerval. Az időzítő beállításait tartalmazza.

old_tv Típusa szintén stuct itimerval. Az időzítő eredeti beállításait tartalmazza. A visszaállításhoz szükséges.

port_in_use BYTE típusú. Azt jelzi, hogy inicializáltuk-e már a soros portot.

Az osztály attribútumait az 5.5. táblázat foglalja össze.

5.5. táblázat. A VM osztály attribútumai

Attribútum	*Típus*	*Tagváltozók*
StackTable	vector<StackInfo>*	BYTE ID LONG tos LONG address LONG size LONG code jmp_buf state
tv	itimerval	–
old_tv	itimerval	–
port_in_use	BYTE	–

VM()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A VM osztály konstruktora. Létrehozza a vermek listáját, beállítja az időzítő értékeit tartalmazó struktúrát. Az időzítő 55 ms-onként fog jelezni, az első jelzés az időzítő indulása után 55 ms-al lesz. Ez a DOS alapértelmezett 18,2 Hz-es órajelét követi. Valamint a `port_in_use` változó értékét is nullára állítja.

StackAlloc(BYTE, LONG, void*)

- *Bemeneti paramétere*: A taszk azonosítója, a verem mérete, a taszk kódjára mutató mutató.
- *Visszatérési értéke*: nincs.

Vermet foglal a taszknak. Az alapértelmezett veremméretet a `config.h` fájlban található `STACK_SIZE` konstans határozza meg. Az eljárás memóriát foglal a megadott mérettel, majd a kapott paraméterek alapján kitölti a taszkokhoz tartozó StackInfo struktúrát. A `state` tagváltozó tárolja el a taszk állapotát. Ezt a `setjmp()` függvényhívással inicializálom. Ez az utasítás a processzor regisztereit menti el a kapott struktúrába. POSIX kompatibilis, tehát hordozható. Párjával, a `longjmp()` hívással együtt fogja megoldani a taszkváltást, így nem lesz szükség annak assembly megvalósítására. Az algoritmus az állapot struktúra utasítás mutatóját a taszk első utasítására állítja, a veremmutatót pedig a most foglalt területre. Végül az így létrehozott struktúrát eltárolja a vermek listájában.

StackFree(BYTE)

- *Bemeneti paramétere*: A taszk azonosítója.
- *Visszatérési értéke*: `NO_SUCH_TASK`, vagy `NO_ERROR`.

A paraméterben kapott taszk vermét szabadítja fel. Ehhez kikeresi a vermet a listában, majd felszabadítja a memóriaterületet. Végül törli a listából a bejegyzést. Ha nem talál ilyen azonosítójú vermet, akkor `NO_SUCH_TASK` üzenettel tér vissza.

RunFirstTask(BYTE)

- *Bemeneti paramétere*: A taszk mutatója.
- *Visszatérési értéke*: nincs.

Az első taszkot futtatja. Az operációs rendszer indulásakor hívódik meg. A paraméterben kapott taszk vermét kikeresi a listából, majd a `longjmp()` hívással betölti a `state` tagváltozóban eltárolt állapotot. Ezzel elkezd futni a taszk. Ha nem talál ilyen taszkot, akkor az operációs rendszer hibával leáll.

CtxSwitch()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A rendszer taszkváltó utasítása. A kernel `PrevRunning` és `Running` attribútuma által mutatott két taszk között vált környezetet. Vagyis elmenti az előzőleg futtatott taszk állapotát, és betölti az újat. Ehhez megkeresi mindkét taszk vermét. A régi állapotát a `setjmp()` utasítással menti el, az újat pedig a `longjmp()`-vel tölti be. Ha visszatöltjük egy taszk állapotát, akkor oda fogunk kerülni a programban, ahol azt elmentettük, tehát jelen esetben az előző utasításra. Azonban a `longjmp()`

második paramétere megadja, hogy ilyen visszatéréskor mit adjon vissza a `setjmp()`. Ha tehát állapot visszatöltéssel kerültünk oda, akkor már nem kell újra állapotot betölteni, hanem egyszerűen vissza kell térni a függvényből. A visszatérések után a betöltött taszk folytatja futását.

ResetTask(BYTE)

- *Bemeneti paramétere*: A taszk azonosítója.
- *Visszatérési értéke*: nincs.

Az azonosítóban kapott taszkot állítja vissza a futása elejére. Ehhez kikeresi az állapotot leíró struktúrát, és annak utasítás mutatóját beállítja a taszk kezdetére.

InstallTickHandler()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A metódus a `signal()` függvénnyel állítja be, hogy milyen függvény hívódjon meg az óraütésekre. Ez jelen esetben a `Tickhandler()` függvény, mely egyik osztálynak sem metódusa, és egyetlen feladata, hogy a Clock osztály `Tick()` metódusát meghívja. Azért definiáltam függvényként, mert a `signal()` függvény ezt várja paraméternek. Az időzítőt a `setitimer()` hívás állítja be a konstruktorban megadott értékekre, és közben elmenti az eredeti beállításokat.

Az időzítéses módszernek vannak előnyei és hátrányai is. Az időzítő ugyanis úgy működik, hogy minden óraütésnél elrontja a processzor egyik szegmensregiszterének értékét, ami hibát generál. Ennek a hibajelzésnek a hatására hívódik meg ezután a beregisztrált kezelőfüggvény. Azonban a hibajelzés csak védett módban tud továbbítódni. A DJGPP viszont DOS-ra épül, tehát amíg DOS-os rendszerhívásokat végez (pl. képernyőre írás), addig a rendszer valós módban működik. Így a jelzésnek meg kell várnia, amíg a processzor visszatér egy ilyen hívásból. Egy másik megoldás lehetne a közvetlenül megszakításból történő taszkváltás, azonban ezt a DJGPP rendszer nem engedi meg, mivel a megszakítást valójában nem a program kezeli le, hanem a DPMI (DOS védett módú interfész) kiszolgáló, amely megszakítás közben a program számára nem követhető módon megváltoztatja a regiszterek értékeit. Így azok visszatöltésekor a rendszer különböző hibákkal leállna.

A módszernek az az előnye, hogy POSIX kompatibilis, azaz UNIX és Linux rendszerekben is meglevő függvényeket használ. Mivel ezek a rendszerek már kizárólag védett módban futnak, ezért ott nincs is hasonló probléma.

A metódus ezen kívül a Ctrl+Break billentyűkombináció lenyomására is telepít egy kezelőfüggvényt (`BreakHandler()`), ami leállítja az operációs rendszert.

ResetTickHandler()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az időzítőt állítja vissza az eredeti értékére.

EnterCritical()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A `cli` assembly utasítással letiltja a megszakításokat.

LeaveCritical()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az `sti` assembly utasítással engedélyezi a megszakításokat.

UARTInit()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A soros portot inicializálja. Beállítja annak sebességét (115200 kbps), valamint az üzenetátviteli paramétereket (8 bites átvitel, paritásellenőrzés nincs, egy stop bit használata). A `port_in_use` attribútum értékét egyre állítja. A soros porttal kapcsolatos funkciókat a `config.h` `UART_ENABLED` konstansával engedélyezhetjük.

UARTSend(BYTE)

- *Bemeneti paramétere*: Elküldendő bájt.
- *Visszatérési értéke*: nincs.

Egy bájtot küld el a soros porton keresztül. A `0x18`-as és `0x19`-es kódú karaktert két bájton küldi el, mivel a tesztelés során kiderült, hogy a `0x18`-as karakter hatására az azt követő bájt duplán kerül elküldésre. A `0x18`-as karater így `[0x19, 0x00]`-ként, míg a `0x19`-es `[0x19, 0x19]`-ként kerül elküldésre.

UARTCheck()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: `MESSAGE_AVAILABLE`, `NO_MESSAGE`.

A soros portot kérdezi le, hogy érkezett-e üzenet. Ennek megfelelően ha van üzenet, akkor `MESSAGE_AVAILABLE` értékkel tér vissza, egyébként pedig `NO_MESSAGE` értékkel.

UARTReceive()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: A kapott bájt.

Egy bájtot olvas a soros portról. Először ellenőrzi, hogy tényleg érkezett-e üzenet (addig vár, amíg meg nem érkezik), majd beolvassa az érkezett bájtot. Ezt utána a visszatérési értékben adja vissza. A `0x18`-as és `0x19`-es karaktert az előbb ismertetett kódolás szerint két bájtban olvassa be.

5.2.6. A Semaphore osztály

Az osztály a taszkok közötti kommunikáció egy formáját valósítja meg, a szemafort. A szemafor segítségével a taszkok erőforrásokat foglalhatnak le, esetleg bevárhatják egymást működés közben. A szemafort egyszerre adott számú taszk foglalhatja le. Az ezután foglalást kérő taszkok felfüggesztődnek, és akkor kapják meg a szemafort, amikor valamelyik foglaló taszk elengedi azt.

Fordítási időben eldönthetjük, hogy szeretnénk-e szemaforokat használni. Ha igen, akkor a `config.h` fájlban hagyjuk meg a `SEMAPHORES_ENABLED` konstanst. Ennek hiányában a fordító ki fogja hagyni a Semaphore objektumot fordításkor.

Attribútumok

value BYTE típusú. A szemafor értékét tárolja. A szemafort még annyi taszk foglalhatja le, amennyi ezen attribútum értéke.

pending A szemaforra várakozó taszkok listája. Taszkvezérlő objektumokra mutató mutatókból álló vektor.

Az osztály attribútumait az 5.6. táblázat foglalja össze.

5.6. táblázat. A Semaphore osztály attribútumai

Attribútum	*Típus*
value	BYTE
pending	vector<TaskP>

Semaphore(BYTE)

- *Bemeneti paramétere*: szemafor kezdeti értéke.
- *Visszatérési értéke*: nincs.

A szemafor osztály konstruktora. Bemeneti paraméterként a szemafor kezdeti értékét kapja meg. Ezt értékül adja a *value* attribútumnak. Az így létrehozott szemafort legfeljebb csak ennyi taszk tudja egyszerre lefoglalni.

Pend(TaskP, LONG)

- *Bemeneti paramétere*: lefoglalást kérő taszk mutatója, várakozási idő (time-out).
- *Visszatérési értéke*: `SEM_ACCEPTED`, `SEM_DENIED`.

A metódus a szemafor lefoglalását végzi. A lefoglalást kérő taszk a paraméterben megadja a taszkvezérlő szerkezetére mutató mutatót, valamint egy időtúllépési értéket. Ha a szemafor értéke nagyobb nullánál, akkor a lefoglalás lehetséges: a szemafor értéke eggyel csökken, a metódus pedig `SEM_ACCEPTED` értékkel tér vissza.

Egyébként a lefoglalási kérelem meghiúsul. A kérő taszk ekkor felkerül a szemafor várakozási listájára. Valamint a szemafor késlelteti a paraméterben kapott időtúllépésnyi idővel. A visszatérési érték ekkor `SEM_DENIED`.

Post()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A szemafor visszaadását végző metódus. Először ellenőrzi, hogy van-e a szemaforra várakozó taszk (a várakozó lista elemszáma nem nulla). Ha van, akkor a lista utolsó taszkját futásra kész állapotba helyezi, majd törli a várakozó listából. A szemafor értékét eggyel növeli. Az így futásra kész állapotba helyezett taszk lefoglalhatja tehát a szemafort, amikor az ütemező megint futtatja.

5.2.7. A Queue osztály

A taszkok közötti kommunikáció egy másik eszköze. Segítségével a taszkok objektumokat cserélhetnek ki egymás között. A sor adott számú objektumot tud tárolni. Egy taszk küldhet objektumot a sorba, vagy elvehet egyet az ott tároltak közül.

A Queue osztály egy sablon osztály, ami azt jelenti, hogy fordítási időben eldönthetjük, hogy milyen típusú objektumok tárolására szeretnénk használni. A rendszerben egyszerre többféle, különböző típust tároló sor is létezhet.

A Queue osztálynál is eldönthetjük fordítási időben, hogy szeretnénk-e használni. Ha nem, akkor töröljük a `config.h` fájlból a `QUEUES_ENABLED` konstanst.

Attribútumok

messages Az üzeneteket tároló vektor. A típusa T, azaz a sablon paramétere. A tényleges típus fordítási időben dől el. Az egyes Queue objektumok egymástól különböző típusokat is tárolhatnak.

pending A Queue objektumra várakozó taszkok vektora. Egy taszk akkor kerül erre a listára, amikor egy üres sorból szeretne olvasni.

Az osztály attribútumait az 5.7. táblázat foglalja össze.

5.7. táblázat. A Queue osztály attribútumai

Attribútum	*Típus*
messages	vector<T>
pending	vector<TaskP>

Queue()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az osztály konstruktora. Üres metódus, nem csinál semmit.

Post(T)

- *Bemeneti paramétere*: az elküldendő adat. Típusa a sablon paramétere.
- *Visszatérési értéke*: nincs.

Egy objektumot küld a sorba. Ha van a sorra várakozó taszk a *pending* vektorban, akkor az annak a végén található taszkot futásra kész állapotba helyezi, valamint törli a vektorból. Ezután a paraméterben kapott objektumot hozzáfűzi az üzeneteket tartalmazó vektorhoz.

Pend(TaskP, T*, LONG)

- *Bemeneti paramétere*: olvasást kérő taszk mutatója, mutató egy megfelelő (T) típusú objektumra, időtúllépés mennyisége.
- *Visszatérési értéke*: `QUEUE_ACCEPTED`, `QUEUE_DENIED`.

A sorból való olvasást végző metódus. Ha a sorban található üzenet (a *messages* attribútum nem üres), akkor a paraméterként kapott T típusú objektum megkapja a sorban tárolt utolsó objektum értékét. Ezután a sorban levő objektum törlődik, és a metódus `QUEUE_ACCEPTED` üzenettel tér vissza.

Ha nincs üzenet a sorban, akkor a kérő taszk felkerül a sorra várakozók listájára. A metódus ezen kívül késlelteti a paraméterben kapott időtúllépési értékkel. A metódus visszatérési értéke ebben az esetben `QUEUE_DENIED`.

5.2.8. A Status osztály

Az osztály a taszkok közötti kommunikáció harmadik formáját valósítja meg. Egy periodikus taszk állapotáról szolgáltat információkat. Egyszerre egy üzenetet tud tárolni, melyet az olvasás nem töröl. Az üzenetet csak a Status objektum tulajdonosa írhatja felül újabbal. Így az objektum használható arra, hogy mindig tudósítson az egyik taszk aktuális állapotáról.

Az osztály a Queue osztályhoz hasonlóan egy sablon osztály. Azaz itt is fordítási időben dől el a tárolt üzenet típusa. A megoldás előnye itt is az, hogy egyszerre több különböző típust kezelő Status típusú objektum is létezhet a rendszerben.

A státusz információ esetében is dönthetünk arról, hogy szeretnénk-e használni. Ezt a `config.h` fájl `STATUS_ENABLED` konstansának megléte illetve hiánya befolyásolja.

Attribútumok

task A tulajdonos taszk vezérlő objektumára mutató mutató.

message Az eltárolt üzenet. Típusa T, azaz ténylegesen fordításkor dől el.

message_stored Azt jelzi, hogy az objektum kapott-e már üzenetet. A nulla érték jelzi, hogy az objektumban még nincs érvényes üzenet. Típusa BYTE.

pending A státusz információra váró taszkok vektora.

Az osztály attribútumait az 5.8. táblázat foglalja össze.

5.8. táblázat. A Status osztály attribútumai

Attribútum	*Típus*
task	TaskP
message	T
message_stored	BYTE
pending	vector<TaskP>

Status(TaskP)

- *Bemeneti paramétere*: A tulajdonos taszk mutatója.
- *Visszatérési értéke*: nincs.

Ez az osztály konstruktora. Feladata az objektum tulajdonosának a beállítása. A metódus egy erre a taszkra mutató mutatót kap paraméterként, amit a *task* attribútumban tárol el. Ezen kívül a *message_stored* attribútum értékét nullára állítja, hiszen az objektum még nem tárol érvényes adatot.

Post(TaskP, T)

- *Bemeneti paramétere*: Az írni kívánó taszk mutatója, a beírni kívánt objektum.
- *Visszatérési értéke*: nincs.

A státusz információ frissítésére szolgáló metódus. Az algoritmus csak akkor fut le, ha az írni kívánó taszk megegyezik a tulajdonos taszkkal. Ha megegyezik, akkor az eltárolt információ felülíródik a most kapottal. Ha a *message_stored* attribútum eddig nulla értékű volt, akkor most egy értékű lesz, jelezve, hogy innentől kezdve érvényes a tárolt információ. Ha volt a státusz információra várakozó taszk a *pending* listában, akkor az abban tárolt összes taszkot futásra kész állapotba helyezi (amelyiknél még nem járt le a várakozási idő), és kiüríti a várakozó listát.

Pend(TaskP, T*, LONG)

- *Bemeneti paramétere*: Az olvasni kívánó taszk mutatója, egy megfelelő (T típusú) objektumra mutató mutató az eredménynek, túllépési idő.
- *Visszatérési értéke*: `STATUS_ACCEPTED`, `STATUS_DENIED`.

A metódus a Status objektumból való olvasásra szolgál. Ha az objektumban érvényes üzenet található (*message_stored* értéke egy), akkor a paraméterben kapott T típusú objektum megkapja az eltárolt objektum értékét, és a metódus visszatérési értéke `STATUS_ACCEPTED` lesz.

Ha még nincs érvényes üzenet az objektumban, akkor a taszk blokkolódik. Vagyis a metódus a paraméterben kapott időtartammal késlelteti, és felveszi az üzenetre várakozó taszkok listájára. Ez esetben a visszatérési érték `STATUS_DENIED`.

5.2.9. A Log osztály

Ez az osztály látja el a naplózással kapcsolatos feladatokat. Egy vektort tárol, ami a rendszerrel kapcsolatos információkat tartalmazza. A vektor string típusú objektumokból áll, melyeknek ezen a szinten nincsen előre meghatározott formátuma. A formátumot a monitorozó állomásnál, és az adatok naplóba helyezésénél kell figyelembe venni. Adatot az operációs rendszer bármelyik objektuma küldhet a naplóba. A felgyülemlett információt az operációs rendszer bizonyos időközönként továbbítja a távoli monitorozó állomásnak.

A Log osztály használata szintén fordítási időben eldönthető. Ha használni szeretnénk, akkor a `config.h` fájlban hagyjuk meg a `LOGGING_ENABLED` konstanst.

Attribútumok

log A napló információit tároló attribútum. Egy string objektumokból álló vektor.

Az osztály attribútumait az 5.9. táblázat mutatja be.

5.9. táblázat. A Log osztály attribútumai

Attribútum	*Típus*
log	vector<string>

Log()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az osztály konstruktora. Tényleges feladata nincsen.

LogInsert(string)

- *Bemeneti paramétere*: új string típusú naplóadat.
- *Visszatérési értéke*: nincs.

A paraméterben kapott új szöveges adatot helyezi el a naplót tároló vektorban. A szöveg formátuma itt nem fontos, a metódus ezt nem ellenőrzi. Ha a napló mérete elért egy adott méretet (ezt a `config.h` fájl `MAX_LOG_SIZE` paramétere állítja), akkor elküldi a napló tartalmát a monitorozó állomásnak.

LogSend()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A napló adatait küldi el a távoli monitorozó állomásnak. Sorban végigmegy az összes eltárolt stringen, melyeket karakterenként elküld a VM objektum `UARTSend()` metódusának meghívásával.

LogReset()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Kiüríti a napló adatait tartalmazó vektort.

5.2.10. Az Actuator osztály

Az Actuator osztály a távoli monitorozó állomás parancsait fogadja, és hajtja végre. Az üzenetek fogadásához a VM osztály soros portot kezelő metódusait használja fel. A kapott üzeneteknek előre definiált formátuma van. Az üzenet első bájtja azonosítja a parancsot. Az üzenet további része pedig a parancs esetleges paramétereit tartalmazza. Az üzenetek hossza változó, a parancstól függ. Az osztály használatát a `config.h` fájl `MONITORING_ENABLED` konstansa állítja.

Attribútumok

Az osztálynak nincsenek attribútumai.

Actuator()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Konstruktor. Valójában egy üres metódus.

ReadCommand()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A metódus eldönti, hogy milyen parancsot kell végrehajtani. Ehhez először ellenőrzi, hogy érkezett-e egyáltalán parancs. Ha igen, akkor annak első bájtját beolvassa. Ezután a kapott érték alapján meghívja a megfelelő parancsot végrehajtó metódust.

StartMonitoring()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Akkor hívódik meg, ha a kapott parancs azonosítója 0 volt. A parancsnak nincsen több paramétere, ezért nem kell többet olvasni a soros portról. A parancs a monitorozást indítja el. A Monitor objektum `enable` attribútumát állítja egybe.

StopMonitoring()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az előző parancs párja. Akkor hívódik meg, ha a kapott parancs azonosítója 1 volt. A monitorozást állítja le.

Filter()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az egyes üzenetek küldését szűri. A parancs azonosítója 2. Egy paramétere van, a szűrendő üzenet azonosítója. A Monitor objektum `filter` tömbjének megfelelő elemét állítja vagy nullába, vagy egybe. Ez irányítja azt, hogy mely üzeneteket küldje el a rendszer a monitorozó állomás felé.

DropTask()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A parancs azonosítója 3. Egy taszk eldobását hajtja végre. Ehhez még egy bájtot olvas a soros portról, ami a taszk azonosítóját fogja jelenteni. Ha az adott taszk éppen fut, akkor félbeszakítja a periódusát, egyébként pedig felfüggeszti.

ChangeCTime()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Egy taszk egyik számítási idejét változtatja meg. Azonosítója 4. A taszk először beolvassa a taszk azonosítóját, majd a számítási idő sorszámát (ez összesen két bájt). Ezután négy bájtban beolvassa az új időt (mivel az LONG típusú), és végrehajtja a módosítást.

ChangePeriod()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Egy taszk periódusidejét változtatja meg. A parancs azonosítója 5. Először a taszk azonosítóját olvassa be, majd négy bájtban az új periódusidőt. Végül végrehajtja a módosítást.

ChangeDeadline()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

A parancs azonosítója 6, és egy taszk határidejét változtatja meg. Az előző parancshoz hasonlóan először a taszk azonosítóját olvassa be, majd négy bájtban az új határidőt. A beolvasás végeztével módosítja a határidőt.

5.2.11. A Monitor osztály

A Monitor osztály a Log osztály felé nyújt egy interfészt. Segítségével olyan üzenetek kerülnek a naplóba, amelyeket a távoli monitorozó állomás képes feldolgozni. Az osztály metódusai az egyes üzenettípusoknak felelnek meg. Az egyes hívások paramétereit az osztály a megfelelő formára alakítva beírja a naplóba. Az Actuator osztályhoz hasonlóan a Monitor osztály használatát is a `MONITORING_ENABLED` konstanssal adhatjuk meg.

Attribútumok

enabled Byte típusú, és azt jelzi, hogy engedélyezve van-e a monitorozás. A metódusok csak akkor írnak a naplóba, ha az értéke 1.

filter Egy 11 elemű BYTE-okból álló tömb. A szűrést valósítja meg. Minden tömbelem megfelel egy üzenetnek. Ha a tömb adott eleme 1, akkor a szűrés érvényes, tehát az az üzenet nem fog bekerülni a naplóba.

A Monitor osztály attribútumait az 5.10. táblázat mutatja be.

5.10. táblázat. A Monitor osztály attribútumai

Attribútum	*Típus*
enabled	BYTE
filter	BYTE[11]

Monitor()

- *Bemeneti paramétere*: nincs.
- *Visszatérési értéke*: nincs.

Az osztály konstruktora. Engedélyezi a monitorozást, és minden szűrést töröl.

TaskCreated(BYTE, BYTE)

- *Bemeneti paramétere*: Létrehozó azonosítója, új taszk azonosítója.
- *Visszatérési értéke*: nincs.

A metódus egy *taszk létrehozva* üzenetet helyez el a naplóban akkor, ha a monitorozás engedélyezve van, és nincs érvényes szűrés az üzenetre. Ehhez először az operációs rendszer idejét bontja fel négy bájtra, majd ezeket bemásolja egy stringbe.

A stringhez fűz továbbá egy üzenetet azonosító bájtot, a taszkot létrehozó taszk azonosítóját (nulla, ha a `main()` hozta létre), és az új taszk azonosítóját. Ezután ezt egy `LogInsert()` hívással elhelyezi a naplóban.

TaskDeleted(BYTE, BYTE)

- *Bemeneti paramétere*: Törlő taszk azonosítója, törölt taszk azonosítója.
- *Visszatérési értéke*: nincs.

A metódus egy *taszk törölve* üzenetet ír a naplóba. Működése megegyezik az előzővel, annyi kivétellel, hogy az idő után a saját üzenet azonosítóját helyezi el, majd a törlő taszk azonosítóját, és végül a törölt taszk azonosítóját.

Semaphore(BYTE, BYTE, BYTE, BYTE)

- *Bemeneti paramétere*: Taszk azonosítója, művelet azonosítója, szemafor azonosítója, szemafor értéke.
- *Visszatérési értéke*: nincs.

Szemafor használatot jelző üzenet. A paraméterben kapott adatokat írja be a naplóba a már ismertetett formában.

IOAccessed(BYTE, BYTE, BYTE, BYTE)

- *Bemeneti paramétere*: Taszk azonosítója, művelet azonosítója, port azonosítója, olvasott/írt üzenet azonosítója.
- *Visszatérési értéke*: nincs.

Egy *ki- ill. beviteli művelet üzenetet* (pl. soros port használat) hoz létre a megismert módon.

IPCSent(BYTE, BYTE)

- *Bemeneti paramétere*: Taszk azonosítója, elküldött üzenet azonosítója.
- *Visszatérési értéke*: nincs.

A metódus egy üzenet küldését jegyzi fel. Az üzenetet a taszk egy sorba, vagy státusz információba is küldheti. A szokásos adatokon kívül a taszk azonosítója és az elküldött üzenet azonosítója kerül be a naplóba.

IPCReceived(BYTE, BYTE)

- *Bemeneti paramétere*: Taszk azonosítója, vett üzenet azonosítója.
- *Visszatérési értéke*: nincs.

Az előző üzenet párja. Egy üzenet vételét írja a naplóba.

StateChanged(BYTE, BYTE, BYTE)

- *Bemeneti paramétere*: Állapot-átmenet azonosítója, taszk azonosítója, számítási mód.
- *Visszatérési értéke*: nincs.

Egy taszk állapotváltozását naplózza. Elsőként az állapot-átmenet azonosítóját küldi el (ez arra utal, hogy a taszk pl. befejezte a futását, felfüggesztődött, stb.), majd a taszk azonosítóját. Végül a használt számítási mód azonosítóját, ám ennek értéke csak akkor releváns, ha a taszk éppen elkezdett futni.

Az állapot átmenet azonosító a következők közül kerülhet ki:

- `STATE_READY_TO_RUN`
- `STATE_RUNNING`
- `STATE_DELAYED`
- `STATE_SUSPENDED`
- `STATE_END_PERIOD`
- `STATE_ABORTED`
- `STATE_DROPPED`

FunctionCalled(BYTE, BYTE, BYTE)

- *Bemeneti paramétere*: Taszk azonosítója, függvény azonosítója, paraméter.
- *Visszatérési értéke*: nincs.

A metódus a taszk egy függvényhívását naplózza. A naplóba kerülő információk a taszk azonosítója, a hívott függvény azonosítója, valamint az esetleges paraméterek értékeit kódoló bájt.

ReturnedFromFunction(BYTE, BYTE, BYTE)

- *Bemeneti paramétere*: Taszk azonosítója, függvény azonosítója, visszatérési érték.
- *Visszatérési értéke*: nincs.

Az elző üzenet párja. A függvényhívásból történő visszatérést írja a naplóba. Az üzenetben szerepel a taszk azonosítója, a függvény azonosítója és a visszatérési értéket kódoló bájt.

6. fejezet

Összefoglalás

Az előzőekben felvázoltam egy valós idejű beágyazott operációs rendszer tervezésének és megvalósításának lépéseit. A fejlesztési folyamat során létrehozott operációs rendszernek a *DynamOS* fantázianevet adtam. Ezzel azonban még nem ér véget a fejlesztés, hiszen annak a tesztelés is szerves része kell, hogy legyen. Az ezzel kapcsolatos módszert a következő alfejezetben ismertetem. Ezt követi a rendszer értékelése. A operációs rendszert elhelyezem a többi rendszer által kifeszített skálán. Felsorolom a rendszer előnyeit és hiányosságait. Végül a lehetséges jövőbeli fejlesztési irányokról nyújtok egy áttekintést.

6.1. Tesztelés

A szoftverfejlesztési folyamat elengedhetetlen része a tesztelés. Ez a munkafázis hoz felszínre számos olyan hibát, melyek a fejlesztés során elsikkadtak, illetve segíti a szoftver minőségének meghatározását. A tesztelést nem a fejlesztési folyamat végén kell elkezdeni, hanem még a fejlesztés közben, hiszen minél később derül fény egy hibára, annál költségesebb lesz annak a kijavítása. Ez feltételezi egy rendszer moduláris felépítését, hiszen ekkor az egyes modulokat külön-külön is tesztelhetjük, nem szükséges az egész rendszernek működnie. A modulok megfelelő tesztelése után egyes modulok együttműködését is vizsgálhatjuk. Végül, ha az egész rendszer elkészült, akkor sor kerülhet az úgynevezett integrációs tesztre, ahol a teljes rendszer együttműködését vizsgáljuk.

A DynamOS fejlesztése során igyekeztem az előbbiekben vázolt tesztelési módszert követni. Először a Kernel és TaskControl objektumokat implementáltam. A két objektumot már tudtam együtt tesztelni. Ellenőriztem, hogy a Kernel objektum helyesen regisztrálja-e a taszkvezérlőket. Megfigyeltem továbbá a különböző taszkvezérlő utasítások hatását is (például felfüggesztés, futásra kész állapotba helyezés, stb.). Az ütemező folyamatos futtatásával pedig az EDF ütemezés helyességét ellenőriztem.

A Clock objektum hozzávételével a határidők kezelését is ellenőrizni tudtam (a Clock objektum `Tick()` metódusa figyeli a határidőket). A Forecast osztály

megvalósításával a rendszer képes lett a többféle számítási mód kezelésére, amit szintén megfelelően ellenőrizni kellett. Ehhez az önálló laboratórium során használt tesztvektorok néhány példányát használtam fel.

A taszkok közti kommunikációt megvalósító osztályokat is lehetett tényleges taszkok futtatása nélkül tesztelni, hiszen ezen osztályok metódusainak a megfelelő paramétereket átadva imitálni lehet egy valós kommunikációs szituációt.

A tesztelés során a legnagyobb kihívást a VM osztály jelentette. Ez az osztály képviseli a hardvert eltakaró virtuális gép réteget. A fejlesztés során sokáig óramegszakításokból történő taszkváltást képzeltem el, melyet assembly nyelven próbáltam megvalósítani. Ez számos nyomonkövethetetlen hibához vezetett, mivel egyetlen hibás utasítás a teljes fejlesztőkörnyezet leállásához vezetett (ezt az óra megszakításkezelő vektorának átállítása okozta). A fejlesztés későbbi szakaszában kiderült, hogy az általam használt DJGPP rendszer egyáltalán nem támogatja a megszakításból történő környezetváltást, így más megoldást kellett találni. Ez az időzítő alapján történő taszkváltás lett. Azonban az assembly nyelvű taszkváltás még ekkor is gondot okozott. A megoldás végül a `setjmp()` és `longjmp()` függvények használata lett, melyek a rendszer összes fontos regiszterének állapotát képesek elmenteni és visszatölteni.

Az így megvalósított VM osztállyal már lehetett a teljes rendszert tesztelni, valóságos taszkokkal. A rendszer tesztelését először két taszkkal végeztem, melyeket váltogatta az ütemező. Ezt követően több taszkkal ellenőriztem az EDF ütemező helyes működését. Majd ismét két taszkkal a számítási idők helyes meghatározását. Végül a kommunikációt ellenőriztem két taszk között.

A rendszert egy 850 MHz-es AMD Duron processzoros számítógépen fejlesztettem és teszteltem, mely 256 MB memóriával rendelkezett. Operációs rendszerként magyar Windows XP SP1-et használtam. A fejlesztői környezet összetétele: DJGPP 2.03, GCC 3.3.3, G++ 3.3.3, GNU Assembler 2.14. Az operációs rendszert MS-DOS 6.22, és a Linux DOS emulátora alatt (a DOSEMU 1.2.10-es, és FreeDOS 1.1.28-as verzió) is kipróbáltam. A rendszer a tesztelés végére elfogadhatóan működött ezeken a konfigurációkon.

A monitorozás teszteléséhez két soros porton összekötött számítógépre volt szükség. Az egyik gépen Windows XP alatt a DynamOS futott, míg a másikon szintén Windows XP alatt az EmMonitor nevű monitorozó program [Bartha04]. A tesztelés során az operációs rendszer taszk létrehozó, állapotváltozást jelölő, illetve kommunikáció használatot jelző üzeneteket küldött, melyeket a monitorozó állomás vett. A tesztelés eredményeképpen meg kellett változtatni a soros portot kezelő eljárásokat, ugyanis a *0x18*-as kódú karakter hatására a következőleg átküldött karakter megduplázódott. Kiderült továbbá, hogy az üzenetek tárolására a string osztály nem alkalmas, ezért a monitornak küldendő üzeneteket nem tároltam el a naplóban, hanem egyből elküldtem a soros porton keresztül.

A monitorozás a tesztelés végére megfelelően működött 115200 baudos sebesség mellett, 8 bites átvitellel.

6.2. Eredmények

A tesztelés során számos hibát sikerült kiszűrni, ám a rendelkezésre álló idő nem tett lehetővé egy minden részletre kiterjedő, alapos tesztelést. Az általam vizsgált példák során azonban az operációs rendszer az EDF algoritmusnak megfelelően ütemezett, a taszkokat helyesen futtatta és állította le, a számítási módokat a használt adaptív algoritmusnak megfelelően rendelte a taszkokhoz. A taszkok közötti kommunikáció is az elvártnak megfelelően működött. A monitorozó eszközzel folytatott kommunikáció szintén sikeres volt, mind az üzenetek küldése, mind a fogadása helyes lett a tesztelés végére.

Az operációs rendszerrel kapcsolatos negatívum a sebesség lehet. A `DEBUG` módot engedélyezve az operációs rendszer minden változáskor kiírja a képernyőre az ütemezési táblázat teljes tartalmát. Ez azonban nem mindig fér bele a rendelkezésre álló egy óraütésnyi időbe. Van amikor a kiírás közben érkezik az óraütés, ami a rendszer leállásához vezethet. A probléma a DJGPP és a DOS működéséből adódik. A DJGPP ugyanis védett módú programokat állít elő, melyek a DOS-ra épülnek. A DOS azonban egy valós módú operációs rendszer. Így minden rendszerhívásnál a processzornak valós módba kell váltania (a képernyőre történő írás pedig DOS rendszerhívással valósul meg), majd vissza védett módra. Egy ilyen váltáshoz el kell menteni a processzor regisztereit, majd a hívás befejeztével újra vissza kell tölteni őket. Ez gyakori rendszerhívásoknál súlyos terhet ró a processzorra. Megoldás lehet egy teljesen 32 bites, védett módú rendszer, mint például a Linux használata. Ez esetben a rendszer már nem fog a két mód között kapcsolgatni.

A feladatkiírásban szereplő webes megjelenítés egyelőre az idő szűkössége miatt nem készült el, ám megvalósítása fontos lenne, mivel a segítségével visszajelzéseket kaphatnék az operációs rendszerrel kapcsolatban, új megvilágításban láthatnám azt.

6.3. Értékelés, tapasztalatok

A diplomatervezés során sikerült létrehoznom egy valós idejű beágyazott operációs rendszert, mely rendelkezik a beágyazott rendszerek alapvető szolgáltatásaival. Operációs rendszer szinten támogatja a többféle feldolgozási szinttel rendelkező taszkok kezelését. Rendelkezik az alapvető taszkkezelő szolgáltatásokkal, és megfelelő eszközkészletet nyújt a taszkok egymás közötti kommunikációjához. A belső állapotait továbbá képes egy külső monitorozó állomás felé továbbítani, majd az ezek alapján javasolt változtatásokat végrehajtani.

Az operációs rendszer rendkívül könnyen portolható, hiszen a használt utasítások nagy része POSIX kompatibilis, azaz Linux és UNIX rendszereken is megtalálhatóak. Ez alól a VM osztály megszakítás tiltó és engedélyező, valamint soros port kezelő metódusai a kivételek. Egy esetleges Linux port megírásakor a `setjmp()` és `longjmp()` függvény által használt `jmp_buf` struktúra tagváltozóira kell még odafigyelni, mivel ezek eltérnek a DJGPP-ben használatosaktól.

A DynamOS operációs rendszer ezen kívül jól konfigurálható. A felhasználó megadhatja, hogy milyen szolgáltatásokra van szüksége, és a lefordított rendszerbe csak ezek fognak bekerülni. Megadható továbbá a taszkok maximális száma, a veremméret és az előretekintő táblázat mérete is.

Az objektum orientált megvalósítás miatt az operációs rendszer biztonságosabb, mint a hagyományos strukturált rendszerek. A taszkok a C++ védelmi mechanizmusai miatt nem változtathatják meg az operációs rendszer belső változóit, valamint csak a számukra elérhető metódusokat képesek meghívni. Így nem is tudnak olyan könnyen hibás állapot előidézni a működésük során.

A rendszer a moduláris felépítés miatt könnyen áttekinthető, és könnyen bővíthető. Az esetleges új osztályokat csak egybe kell fordítani a már meglevőkkel, és máris lehet használni őket a taszkokban, vagy magában a kernelben.

Az operációs rendszer hiányosságai közé tartozik, hogy egyelőre csak a kemény valós idejű taszkok futtatását támogatja. Sem puha-, sem nem valós idejű taszkokat nem tudunk futtatni segítségével. Az operációs rendszert ezen kívül nem lehet önállóan használni, mindenképpen szükséges egy gazda operációs rendszer, ami fölött fut. Hiányosság továbbá, hogy az óraütések nem feltétlenül akkor jutnak érvényre, amikor azok tényleg megtörténnek. Ezt a már említett DPMI (DOS védett módú interfész) működése okozza.

Az ütemezési algoritmus is rendelkezik hiányosságokkal. Az egyik problémája az, hogy ha egy taszk nem fér be az ütemezésbe, akkor egyszerre csak egy másik taszk számítási idejét tudja csökkenteni. Így ha egy csökkentéssel nem szabadítható fel elég erőforrás, akkor a szóban forgó taszk el lesz dobva. Az ütemező ezen kívül nem figyeli a taszkok egymástól való függését. Például nem kezeli eltérő módon azt a helyzetet, amikor egy taszk egy másik eredményére vár.

Az operációs rendszer felépítésében leginkább a μC/OS-re és a Hartik-ra hasonlít. Azokhoz hasonlóan ez is egy kis méretű rendszer, alapvető szolgáltatásokkal. Ez megfelel az eredetileg kitűzött céloknak. Erőforrásigénye azonban a μC/OS-énél jóval nagyobb. Az adaptív működéshez szükséges többlet memória és számítási idő elég jelentős. Az operációs rendszer lefordított mérete majdnem nyolcszorosa a μC/OS-ének (nagyjából 300 KB). Ez többek között az objektum orientált megvalósításnak, illetve a kisebb mértékű optimalizációnak köszönhető. Az erőforrás igény és a szolgáltatások száma alatta marad a megismert nagyobb operációs rendszerekének, mint például az rtems vagy a Fiasco.

Személyes tapasztalatom a fejlesztéssel kapcsolatban az, hogy a diplomatervezéshez rendelkezésre álló idő kevés egy éles helyzetben is alkalmazható rendszer létrehozásához. Ehhez még hosszas tesztelésre és (leginkább teljesítménybeli) elemzésre lenne szükség. Azonban a rendszer az eredeti célként kitűzött kísérleti rendszerként megállja a helyét.

A választott objektum-orientált fejlesztési módszertan részben megfelelő volt, részben nem. Az objektum orientáltság átláthatóbbá, bővíthetőbbé, sokoldalúbbá (lásd például a többféle objektumot tárolni képes kommunikációs objektumokat) tette a rendszert. Ugyanakkor megnövelte az erőforrásigényét mind a számítási kapacitás, mind a memória felhasználás szempontjából. Úgy gondolom azonban, hogy

egyrészt a beágyazott rendszerek is egyre nagyobb erőforrásokkal fognak rendelkezni, másrészt a teljesítményen egy optimalizálással még javítani lehet.

A DJGPP, mint fejlesztői környezet nem a legideálisabb választás volt, mint az a fejlesztés során kiderült. Főként a DOS alapúság róható fel a hiányosságaként. Ezzel kapcsolatban már többször említettem a sebesség és időzítésbeli gondokat. Olyan szempontból azonban jó, hogy nagyrészt POSIX kompatibilis, tehát egy tisztán GCC port megírása nagyon egyszerű, sőt a két platformon akár ugyanazt a forráskódot is le lehet fordítani megfelelő előfordítói utasításokkal (pl. `#ifdef`).

Összefoglalásképpen a létrehozott operációs rendszer egy kis méretű, alapvető szolgáltatásokkal bíró, ugyanakkor futás közbeni adaptivitásra képes rendszer. Felhasználási területe leginkább a kis, vagy közepes méretű beágyazott alkalmazások körében keresendő, ahol az alkalmazásoknak csak az alapvető operációs rendszer szolgáltatásokra van szükségük. Célszerű az operációs rendszert olyan környezetben alkalmazni, ahol a változó környezeti feltételek miatt előfordulnak túlterhelt rendszerállapotok. Ekkor látszanak ugyanis igazán az adaptív ütemezési algoritmus előnyei. Ehhez azonban olyan alkalmazások futtatása szükséges, melyek képesek többféle futási mód felkínálására.

6.4. Továbbfejlesztési lehetőségek

Az operációs rendszer fejlesztésének első szakasza ezzel lezárult, azonban még további megoldandó feladatok maradnak vele kapcsolatban.

Az első feladat az operációs rendszer honlapjának elkészítése. Innen szabadon elérhető lenne a rendszer forráskódja, így azt bárki véleményezni tudná, esetleg javaslatokat fogalmazhatna meg vele kapcsolatban.

Talán a legfontosabb hátralevő feladat az alapos tesztelés, amire mindenképpen szükség van, hiszen egy beágyazott operációs rendszer esetén alapvető szempont a megbízható működés. A monitorozó programmal való együttműködés szintén további tesztelésre szorul.

Fontos lenne továbbá a Linux illetve UNIX port elkészítése, amely kiküszöbölné a DOS platform hiányosságait. Ez ugyanakkor szélesebb körben is alkalmazhatóvá tenné a rendszert.

Mint már említettem, a rendszer csak a kemény valós idejű taszkokat kezeli. Ezt ki kéne bővíteni a puha valós idejű, a szporadikus (az ilyen taszkoknál nincs egy adott periódusidő kikötve, csak egy minimális időintervallum van definiálva két futásuk között) és a nem valós idejű taszkokkal. E taszkok kezelésére az ütemező algoritmust is fel kéne készíteni.

További feladat még a rendszer optimalizálása mind memória felhasználás, mind számítási idő szempontjából. Az algoritmusok egyszerűsítésével, racionalizálásával, és az attribútumok takarékosabb megválasztásával még több erőforrást lehetne az alkalmazások számára hagyni.

A fenti feladatok elvégzése után további meghajtókat lehetne írni a rendszerhez, például egy Ethernetes kommunikációkezelőt. További ütemezési algoritmusokat is

lehetne implementálni, melyek közül a felhasználó választhatna. Végül az operációs rendszert függetleníteni kéne más operációs rendszerektől (DOS, Linux), vagyis önállóan futtathatóvá kéne tenni. Ez a nem PC-s felhasználási környezethez ugyanis elengedhetetlen.

Irodalomjegyzék

Könyvek

[Buttazzo02] Buttazzo, Giorgio C., *„Hard Real-Time Computing Systems”*, Kluwer Academic Publishers, Norwell, 2002.

[KLSz99] Kondorosi, K., László, Z., Szirmay-Kalos, L., *„Objektum-orientált szoftverfejlesztés”*, ComputerBooks, Budapest, 175–195. o., 1999.

Disszertációk, diplomatervek

[Bartha04] Bartha, S., *„Beágyazott információs rendszerek monitorozása”*, diplomaterv, BME MIT, Budapest, 2004.

Cikkek, konferenciaanyagok

[BuAb02] Buttazzo, G., Abeni, L., *„Adaptive Workload Management through Elastic Scheduling”*, Real-Time Systems, 23, 7-24. o., 2002.

[CJCP00] Cardei, I., Jha, R., cardei, M., Pavan, A., *„Hierarchical Architecture For Real-Time Adaptive Resource Management”*, The IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, New York, USA, 2000.

[DVJGS99] Doerr, B. S., Venturella, T., Jha, R., Gill, C. D., Schmidt, D. C., *„Adaptive Scheduling for Real-time, Embedded Information Systems”*, Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC), St. Louis, Missouri, October 24-29, 1999.

[GKSC02] Gill, C. D., Kuhns, F., Schmidt, D. C., Cytron, R. K., *„Empirical Differences between COTS Middleware Scheduling Strategies”*, Proceedings of the Distributed Objects and Applications (DOA) conference, Irvine, CA, October/November, 2002.

[GLCA02] Buttazzo, G. C., Lipari, G., Caccamo, M., Abeni, L., *„Elastic Scheduling for Flexible Workload Management"*, IEEE Transactions on Computers, Vol. 51., NO. 3., 2002.

[NBGG02] Neema, S., Bapty, T., Gray, J., Gokhale, A., *„Generators for Synthesis of QoS Adaptation in Distributed Real-Time Embedded Systems"*, First ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002), Pittsburgh, PA, October 6-8, 2002, pp. 236-251.

[SLCB00] Sha, L., Liu, X., Caccamo, M., Buttazzo, G., *„Online Control Optimization Using Load Driven Scheduling"*, Proceedings of the 39th IEEE Conference on Decision and Control, Sydney, Australia, December 2000.

[ZiLoSh02] Zinky, J., Loyall, J., Shapiro, r., *„Runtime Performance Modeling and Measurement of Adaptive Distributed Object Applications"*, Proceeding of International Symposium on Distributed Object and Applications, DOA 2002, October 28-30 2002, University of California, Irvine CA USA.

Internet

[DJGPP] `http://www.delorie.com/djgpp`, 2004. február 17., 16:14

[DROPS] `http://os.inf.tu-dresden.de/drops/`, 2004. február 25., 9:58

[Eros] `http://www.eros-os.org/`, 2004. február 16., 12:20

[Fiasco] `http://os.inf.tu-dresden.de/fiasco/`, 2004. február 16., 14:30

[Google] `http://directory.google.com/Top/Computers/Software/Operating_Systems/Realtime/Open_Source/`, 2004. február 16., 11:34

[Hartik] `http://hartik.sssup.it/`, 2004. január 28., 10:40

[JBed] `http://www.esmertec.com`, 2004. március 4., 16:10

[proc] `http://www.nilsenelektronikk.no/`, 2004. február 16., 11:42

[RTEMS] `http://www.rtems.com/`, 2004. február 16., 14:45

[RTJava] `http://www.rtj.org`, 2004. március 4., 15:35

[rtmk] `http://rtmk.sourceforge.net/`, 2004. február 16., 15:15

[SHaRK] `http://shark.sssup.it/`, 2004. január 28., 11:10

[Tiny] `http://webs.cs.berkeley.edu/tos/`, 2004. február 25., 10:56

[uCOS] `http://www.ucos-ii.com/`, 2004. február 16., 11:50

Függelék

F.1. Fordítási útmutató

A DynamOS fordításához rendelkeznünk kell valamilyen DOS-os környezettel. Ez lehet az MS-DOS valamelyik verziója, valamelyik Windows verzió, vagy akár a Linux DOSEMU emulátora.

Ha ez megvan, akkor le kell töltenünk, és telepítenünk kell a DJGPP rendszert. Ehhez látogassunk el az alábbi címre (vagy használjuk a mellékelt CD-n található fájlokat): `http://www.delorie.com/djgpp/zip-picker.html`. Az oldalon először válasszunk egy nekünk megfelelő FTP kiszolgálót. A következő listából válasszuk a *Build and run programs with DJGPP* lehetőséget, majd ezt követően az általunk használt operációs rendszert. A használni kívánt programozási nyelvek kiválasztásánál válasszuk a C-t, a C++-t és az Assembler-t. Ha szeretnénk integrált felhasználói felületet (RHIDE), akkor válasszuk ki azt a lehetőséget is. Ha mindez megvan, kattintsunk a *Tell me which files I need* gombra! Az erre megjelenő oldalon töltsük le az összes letöltésre felkínált fájlt.

A fájlok alatt találunk egy rövid angol nyelvű telepítési útmutatót, ennek lépéseit hajtsuk végre. MS-DOS alatt figyelni kell arra, hogy a DJGPP rendszernek szüksége van a hosszú fájlnevek támogatására. Mivel az MS-DOS ezt nem támogatja, egy ezt megoldó programot is le kell töltenünk (szintén megtalálható a mellékelt CD-n). Egy ilyet találhatunk például a következő címen: `http://www-user.tu-chemnitz.de/~heha/hs_freeware/freew.html`, ahol a DOSLFN nevű programot kell letölteni. Továbbá olyan kitömörítő programot használjunk, ami szintén támogatja a hosszú fájlneveket.

A DJGPP telepítése után már lefordíthatjuk a DynamOS-t. Ezt a legegyszerűbben a mellékelt Makefile segítségével tehetjük meg. Egyszerűen adjuk ki a `make` parancsot a program könyvtárában, és a GNU Make segédprogram lefordítja a rendszert.

Ha az operációs rendszerrel a saját alkalmazásunkat szeretnénk lefordítani, akkor a Makefile `APP` változóját írjuk át a saját alkalmazásunk fájlnevére (kiterjesztés nélkül). Ennek hatására az operációs rendszer már a mi alkalmazásunkkal fog lefordulni.

A fordítás másik módszere az RHIDE felület használata. Ehhez hozzunk létre egy új projektet a *Project* menü *Open project* parancsával. A létrehozott projekthez adjuk hozzá az *Insert* billentyű lenyomásával az operációs rendszer C++ forrásfájljait. Ha saját alkalmazást akarunk használni, akkor azt is adjuk a projekthez. Ezután az *F9* billentyűvel lefordíthatjuk, a *Ctrl+F9* billentyűkombinációval pedig futtathatjuk az operációs rendszert.

F.2. Alkalmazás fejlesztői útmutató

A DynamOS-hez írt alkalmazásokat C++ nyelven készíthetjük el. Az alkalmazások kódja mellé a `main()` függvényt is szükséges elhelyeznünk.

A fájlt kezdjük a szükséges fejlécek beillesztésével:

```
#ifndef _INCLUDES_H
#include "includes.h"
#endif
```

Ez szükséges az operációs rendszer szolgáltatások használatához. Ezután helyezzük el azokat a könyvtár hivatkozásokat, melyeket használni szeretnénk (pl. `math.h` a matematikai funkciókhoz, `iostream` a képernyőre íráshoz). Ezt követően definiáljuk az operációs rendszer használni kívánt objektumait külső objektumokként:

```
extern VM vm;
extern Kernel kernel;
#ifdef MONITORING_ENABLED
extern Monitor monitor;
#endif
```

A kernel objektumra mindenképpen szükség van a taszkok regisztráláshoz és az operációs rendszer elindításához. A vm objektumra akkor van szükség, ha engedélyeztük a soros port kezelését, és az alkalmazásunk használni is szeretné azt. A monitor objektum pedig akkor kell, ha a taszkokkal történő eseményeket szeretnénk monitorozni a külső monitorozó állomás segítségével.

Ezt követően soroljuk fel az általunk használni kívánt objektumokat. Minden taszkhoz szükség van egy taszkvezérlő objektumra. Ugyanígy globális objektumként definiáljuk az összes használni kívánt kommunikációs objektumot. Például:

```
TaskP task1;
Semaphore sem;
Queue<LONG> queue;
Status<string> status;
```

A TaskP típus a TaskControl*-ot takarja. A taszkvezérlőket azért célszerű mutatóként deklarálni, hogy később a `main()` függvényben megfelelő módon hívhassuk meg a konstruktorukat. A taszkokban innentől kezdve hivatkozhatunk ezekre az objektumokra is.

Most a taszkok kódjai következnek, globális függvényként definiálva. Egy taszk csontváza a következőképpen néz ki:

```
void task_1(void)
{
  BYTE c;
  for (;;)
```

```
  {
    c = task1->GetCMode();
    if (c == 0)
    {
    }
    if (c == 1)
    {
    }
    task1->EndRun();
  }
}
```

A taszk tehát egy végtelen ciklusban fut. A ciklus elején lekérdezi a futási módot, és azt eltárolja egy lokális változóban. Ezután a különböző futási módokat `if` utasításokkal különíthetjük el egymástól. A taszk kódjában bármilyen C, illetve C++ függvényhívás használható. A taszk eldobásával nem kell törődnünk, ezt az esetet az operációs rendszer elintézi helyettünk. A ciklus végén hívjuk meg a taszknak megfelelő taszkvezérlő objektum `EndRun()` metódusát. Ezzel a taszk visszaadja a processzort a kernelnek, és legközelebb már csak a következő periódussal fog futni, mégpedig a ciklus elejéről.

A taszkok kódja után következik a `main()` függvény, aminek a C++ konvenciók miatt int típusú visszatérési értékkel kell rendelkeznie. A függvény elején hozzuk létre az egyes taszkokhoz tartozó számítási időket tartalmazó listákat, majd helyezzük el benne a számítási időket:

```
  list<LONG> t1_;

  t1_.insert(t1_.begin(), 1);
  t1_.insert(t1_.begin(), 5);
```

A számítási idők itt az operációs rendszer óraütésében értendők. Egy egység 55 ms-nak felel meg. A számítási idők sorrendje lényegtelen, a taszk létrehozásánál a rendszer csökkenő sorrendbe rendezi őket. A listába írásra használhatjuk az `insert()`, `push_back()`, vagy `push_front()` metódusokat.

Ezt követően hozzuk létre a taszkvezérlő objektumot az alábbi módon:

```
  task1 = new TaskControl(1,(void *)task_1, 7, 8, &t1_, 0);
```

A konstruktor paraméterei sorrendben a következők: a taszk azonosítója, a taszk kódjára mutató mutató (a taszknak megfelelő globális függvény nevét írjuk ide), a taszk relatív határideje, a taszk periódusideje, mutató a taszk számítási ideit tartalmazó listára, és végül a taszk késleltetése (azaz mikor legyen először futásra kész a taszk).

Most már készen állunk a taszk beregisztrálására:

```
  kernel.RegisterTask(task1);
```

Ezt végezzük el minden taszkra, majd indítsuk el az operációs rendszert:

```
kernel.Start();
```

Innentől kezdve már nincs más feladatunk. Ha a program ide ért a futásban, akkor többé már nem fog ide visszatérni, hiszen a taszkokat kezdi el futtatni. Így lezárhatjuk a `main()` függvényt.

A taszkunk kódjában használhatjuk a deklarált kommunikációs objektumokat. Egy sorba írás például a következőképpen néz ki:

```
queue.Post(value);
```

Itt a `queue` objektum egy sor, a `value` pedig a taszk valamilyen belső változója. Ezt a következőképpen olvashatjuk ki egy másik taszkból:

```
queue.Pend(task, &value, 100);
```

Itt `queue` az előbb használt sor objektum. A `task` változó a olvasó taszk vezérlőjére mutat. A `value` itt is a taszk egy belső változója. Az olvasáshoz ennek a címét kell átadni, mivel ebben a változóban kapjuk meg az eredményt. Az utolsó paraméter az időtúllépés értéke. Ha a sor üres, akkor a taszk ennyi ideig fog várakozni.

A többi kommunikációs objektum kezelése nem tér el nagyban a sorétól. Az azoknál alkalmazott utasítások paraméterezése megtalálható az osztályok leírásánál. A diplomatervhez mellékelt CD-n több példaprogram is megtalálható, melyek bemutatják ezek használatát.

A monitorozáshoz csak a megfelelő üzenetet generáló metódusokat kell meghívnunk. Egy sorba írás esemény például a következő utasítással küldhető el:

```
#ifdef MONITORING_ENABLED
   monitor.IPCSent(1, 1);
#endif
```

Itt az első paraméter a taszk, a második pedig az üzenet azonosítója. A sorból olvasás esemény elküldése pedig az alábbi módon történhet:

```
#ifdef MONITORING_ENABLED
  monitor.IPCReceived(2, 1);
#endif
```

A Monitor osztály leírásánál (5.2.11. fejezet) megtalálhatóak a monitorozó állomásnak küldhető üzenetek, és a szükséges paramétereik. Az üzenetek elküldéséhez elég a monitor objektum metódusait meghívni.

F.3. A CD-melléklet tartalma

Jelen diplomaterv mellé egy CD melléklet is tartozik, mely a diplomatervet, az operációs rendszer forráskódját, és a lefordításhoz szükséges eszközöket tartalmazza.

A CD lemez tartalma a következő:

- **Diplomaterv könyvtár**: A diplomaterv elektronikus változata. A `PDF` alkönyvtárban a PDF formátumú, a `PS` alkönyvtárban pedig a PS formátumú változat található meg.
- **DJGPP könyvtár**: Az operációs rendszer fordításához szükséges DJGPP fejlesztőkörnyezet 2.03-as változata található ebben a könyvtárban. Az angol nyelvű telepítési útmutató megtalálható a `readme.1st` fájlban.
    - **DOSLFN alkönyvtár**: A DJGPP használatához MS-DOS alatt szükség van a hosszú fájlnevek támogatására. Ebben a könyvtárban egy ezt lehetővé tevő meghajtóprogram található.
- **DynamOS könyvtár**: Ebben a könyvtárban található a DynamOS forrása, és néhány lefordított példaalkalmazás.
    - **Binary alkönyvtár**:
        - **Example1 alkönyvtár**: Az első példaalkalmazás lefordított, futtatható változata. Ebben hat taszk fut, melyből kettő periodikus. E kettőnek kétféle számítási módja van, a többi nem periodikusnak pedig egy. A taszkok a futásuk során a képernyőre írják az azonosítójukat, az aktuális rendszeridőt, valamint azt, hogy melyik számítási módban futnak. A példa az ütemezési képességeket demonstrálja.
        - **Example2 alkönyvtár**: Ebben az alkalmazásban három taszk szerepel. Mindhárom egyféle számítási módot használ. Az első két taszk egy lokális változó értékét növeli folyamatosan, melyet egy sorban helyez el. Az elküldött értékeket kiírják a képernyőre. A harmadik taszk a két sorban található értéket kiolvassa, összeadja, majd kiírja a képernyőre. Ez a példa a taszkok közötti kommunikációt szemlélteti.
        - **Example3 alkönyvtár**: Ebben a példában két taszk szerepel, mindkettő két-két futási móddal. Az első taszk az alapértelmezett futási módban egy lokális változó értékét növeli, majd elküldi egy sorba. Csökkentett módban nem növeli az értéket, csak elküldi. A második taszk a sorból kiolvassa az előbb elküldött értéket, és az első futási módban megnöveli eggyel, majd kiírja. A csökkentett módban ez a taszk sem növel, csak kiírja a képernyőre az értéket. A taszkok futási paraméterei miatt csak az egyik futhat az alapértelmezett futási módjában. A példa a különböző futási módok használatát szemlélteti.

- **Example4 alkönyvtár**: A negyedik példában ismét két taszk szerepel, most egyféle futási móddal. Az első taszk nagyobb periódusidővel rendelkezik, és a futás során egy változó értékét növeli öttel, majd belerakja egy státusz objektumba. A második taszk kisebb periódusidővel rendelkezik, ezért egy új adat beérkezése között többször is kiolvassa a státusz objektum értékét. A kapott eredményt kiírja a képernyőre. A példa a státusz objektum használatát demonstrálja.
- **Example5 alkönyvtár**: Az ötödik példa alkalmazásban két taszk szerepel, ám ezek közül csak egyet reigsztrál a `main()` függvény. Ennek a taszknak mindössze annyi a feladata, hogy minden futása során hozza létre és regisztrálja a második taszkot, majd töröltesse azt a rendszerrel. A második taszk nem periodikus, és mindössze annyit csinál, hogy a képernyőre írja, hogy fut. A példa a futás közbeni taszkregisztrációt szemlélteti.

◦ **Source alkönyvtár**: Az operációs rendszer, és a példaprogramok forrásai találhatóak ebben a könyvtárban.

- **DynamOS alkönyvtár**: Itt található az operációs rendszer forráskódja. Az operációs rendszer mellett található alkalmazás kód csak az operációs rendszert indítja el, nem hoz létre egy taszkot sem. A fordítás a `make` paranccsal végezhető el.
- **Example1 ... 5 alkönyvtár**: A fent ismertetett példaalkalmazások kódjai, és a megfelelő makefájlok találhatóak ezekben a könyvtárakban. A lefordításukhoz másoljuk őket az operációs rendszer forrása mellé, majd adjuk ki a `make` parancsot.