tanulmányok

119/1981

MTA Számítástechnikai és Automatizálási Kutató Intézet Budapest

MAGYAR TUDOMÁNYOS AKADÉMIA
SZÁMITÁSTECHNIKAI ÉS AUTOMATIZÁLÁSI KUTATO INTÉZETE

- Ó fiam Mihók, édes Mihókom, nem a boglyában kellett volna tüt szurnod, hanem a kalapodba!
- Csakugyan! Majd máskor ugy teszek!

(Székely népmese)

REAL-TIME PROGRAMRENDSZEREK ESEMÉNYVEZÉRELT SZERVEZÉSE (Esettanulmány a Péti Nitrogénmüvek adatgyüjtő rendszeréről)

Irta:

Sztanó Tamás

Tanulmányok 119/1981.

A kiadásért felelős:

DR VÁMOS TIBOR

ISBN 963 311 116 1
ISSN 0324-2951

Készült a
KSH Nemzetközi Számítástechnikai Oktató és Tájékoztató
Központ Reprográfiai Üzemében
1981/116.

TARTALOMJEGYZÉK

Oldal

1. BEVEZETÉS

A real-time rendszerekkel kapcsolatos software (és hardware) kérdések világviszonylatban az érdeklődés középpontjában állnak. Ennek ellenére számos alapvető kérdés vár még válaszra. Ezek közül csak kettőt emelünk ki:

- a "felhasználó-központu", "testreszabott", alkalmazkodóképes, de ugyanakkor megbizható, ujra felhasználható komponensekből álló software kérdését, és
- a párhuzamos müködésü programrendszerek kidolgozásának, "poloskátlanitásának" metódusait, segédeszközeit.

Az a software, amit a Folytonos Folyamatok Osztályán a Péti Nitrogénmüveknél üzembeállitott R-10-es számitógépre kidolgoztunk, célkitüzéseiben és megoldásaiban szintén sok ponton kapcsolódik az emlitett kérdéskörhöz. Az alábbiakban az e munkával kapcsolatos tapasztalatainkról számolunk be, arról, hogy mit hogyan oldottunk meg, mi vált be, mi nem, minek éreztük hiányát, mit tartunk általánosithatónak, mit módositandónak és mik a további elképzeléseink vele kapcsolatban.

A PN-be telepitett mérésadatgyüjtő és feldolgozó rendszer feladatait és a vele szemben támasztott követelményeket a rendszerterv és a munka zárójelentése részletesen tartalmazza. Ezen a helyen inkább csak az általunk kidolgozott software strukturájáról és müködésének szervezésről lesz szó, mintsem a rendszer által ténylegesen ellátott feladatokról.

Ugy véljük, hogy a tulnyomórészt R-10 assembler nyelven megirt rendszerben alkalmazott megoldások nem szigoruan ehhez a géphez és ehhez a feladathoz kötődnek, jól általánosithatók és igy kiindulópontját jelenthetik egy hasonló célu software más gépen történő vagy akár gépfüggetlen megvalósitásának.

2. A RENDSZER FŐ JELLEMZŐI

A rendszer tervezésénél két lényeges szempontot tartottunk szem előtt. Az egyik a mobilitás, a másik az általánosság. Mobilitást követelt meg már a feladatkiirás, amelynek alapján néhány "biztosan szükséges" funkción tul inkább csak a "szóbajöhető" funkciók köre volt többé-kevésbé behatárolható (ez a szituáció egyébként alighanem inkább tipikus, mintsem különleges). Mobilitást kiván az üzemi élet, amely nap mint nap ujabb problémák megoldását követeli meg az emberektől csakugy, mint az őket kiszolgáló számitógéptől.

A felhasználói szempontok mellett szem előtt kellett tartanunk a kidolgozói szempontjainkat, amelyek a munka gazdaságossága érdekében megkövetelték az általánosságra való törekvést, azt, hogy a munka - az egyszeri felhasználáson tulmenően - ne csak a fejekben összegyült tapasztalatok "gazdagodását", hanem általánosabban használható eszközöket, ujra felhasználható megoldásokat eredményezzen.

Ezen kettős törekvésnek eredményeképpen alakult ki egyrészt az a fogalomrendszer, amely magában foglalja mindazt, ami az adott megoldásban a konkrét alkalmazástól és konkrét géptől független. (Ide tartozik a feladat dekomponálásának módja, a rendszer felhasználói szintü kialakitásának, használatának eszközei, szabványos eljárások, konvenciók, stb., mint azt majd látni fogjuk.)

Másrészt kialakult a felhasználói szintü makro-tevékenységeket ellátó elemeknek ("müveleteknek") egy olyan készlete, amelytől azt reméljük, hogy segitségével a konkrét alkalmazásnál eddig és ezután felmerülő igények nagyrésze kielégithető, és hogy elemei - kisebb nagyobb mértékben más alkalmazásoknál szintén ujra felhasználhatók.

A "felhasználói müveleteknek" ezen készletével kapcsolatban utalunk a rendszer dokumentációjára. Jelen beszámolóban ezzel bővebben nem foglalkozunk.

Annak ellenére, hogy a kidolgozott software assembler nyelvü, a rendszer tervezésénél-kidolgozásánál erősen befolyásolt bennünket a SIMULA nyelv fogalomrendszere. Ezen nyelv lehetővé teszi rendszerünk struktúrájának és müködésének teljes részletességü leirását oly annyira, hogy a munka befejezése óta elkészült egy a rendszer továbbfejlesztett változatát szimuláló SIMULA nyelvü program.

Jelen tanulmány illusztrációs célra ezen programból tartalmaz részleteket.

3. A TASZK FOGALOM

Az általunk kialakitott szervezés szerint a gép "felhasználói szinten látható" tevékenységét egymással párhuzamosan végrehajtásra kerülő, egymással együttmüködő programok (taszkok) müködése eredményezi. Minden taszknak megfelel egy, a felhasználó szempontjából oszthatatlan akció és erre mindannyiszor sor kerül, valahányszor a taszk müködési feltételei (ujból) teljesülnek. Müködési feltételeinek teljesültéig a taszk passziv állapotban van.

A taszk müködését (akcióinak sorozatát) a felhasználó az illető taszk létrehozásával indithatja el, ill. megsemmisitésével állithatja le.

Egy-egy taszk látja el pl. bizonyos kiválasztott üzemi jellemzők visszamenőleges értékeinek kigyüjtését és kilistázását, vagy egy kiválasztott készülék stacionárius állapotának ellenőrzését, avagy egy üzemrész adott periódusra vonatkozó anyagmérlegének összeállitását.

Ugy véljük, hogy a felhasználó számára ez a szervezés teljesen természetes, mivel ilymódon a software strukturájában visszatükröződhetnek a géppel kapcsolatban álló külső rendszer párhuzamos folyamatai.

Ezt a strukturát az tette lehetővé, hogy a rendelkezésünkre álló operációs rendszer (PCM) biztositotta nagyszámu (max. 127) on-line program, valamint egy háttérprogram párhuzamos futását, valamint biztositotta a párhuzamos programok szinkronizációjához szükséges alapvető eszközöket, um. az esemény- és erőforráskezelést.

A taszkok maguk és a taszkok együttmüködésének szervezése operációs rendszertől független abban az értelemben, hogy ezeken belül az operációs rendszer sajátosságait nem használtuk ki.

Mindössze annyit tételeztünk fel, hogy létezik egy olyan "futtató rendszer", amely a taszkok között a gép hardware erőforrásait (memória, processor, stb.) valamilyen stratégia szerint megosztja, továbbá az általunk definiált software erőforrások és események kezelését elvégzi.

Az alábbiakban taszkjaink jellegzetességeit és az ennek megfelelő programstrukturát ismertetjük.

3.1 Paraméterezhetőség

Az egyes taszkok programjai - az általánosság, ujrafelhasználhatóság érdekében - olyan "absztrakt" funkciókat irnak le, mint pl. "adatgyüjtés", "stacionaritás ellenőrzés", stb. A taszk-funkciók konkretizálása, adatkapcsolatainak és egyéb müködési jellemzőinek megadása paraméterezéssel történik. Előző példánkban pl. paraméterként lehet megadni a feldolgozandó üzemi jellemzők azonositóit, a stacionaritás ellenőrzésénél a megadott "türés" mértékét stb.

Más szóval ez azt jelenti, hogy az egyes taszkok programja - a SIMULA osztálydeklarációnak megfelelően - csak az illető taszk prototipusát határozza meg. A müködő taszk példányok létrehozása ezen prototipus alapján történik a paraméterek lerögzitésével.

Taszk példányok létrehozására, ill. müködő taszk-példányok megsemmitésére a rendszer életének bármely pillanatában sor kerülhet.

3.2 Emlékezet

Egy-egy taszk-akció kimenete nem csupán (aktuális) bemenetétől függhet, hanem függhet az akciót végrehajtó taszkpéldány belső változóinak a korábbi akciók eredményeként kialakult állapotától is. Pl. taszk formájában fogalmazható meg egy adaptiv

szabályozó algoritmus, ami minden egyes akciója alkalmával a szabályozáson tulmenően saját paramétereit is korrigálja.

3.3 Függetlenség

Az egyes taszkok egymástól függetlenek abban az értelemben, hogy egy-egy akciójuk eredménye csak a bementektől és a belső változóik aktuális állapotától függ, környezetüktől nem. A környezettől csak az egyes taszk-akciók végrehajtási ideje függ, de ez az időtartam is garantáltan véges, ha mind a taszk, mind pedig környezete "korrekt". (A taszkok korrektségét később részletezendő programozási konvenciók biztositják.)

A taszkok ilyen értelmü függetlenségét mind a rendszer áttekinthetősége, tesztelhetősége, mind pedig a kidolgozáshoz szükséges munkamegosztás egyaránt megkivánta. A taszkok ily módon egyenként, a többi taszktól függetlenül szekvenciális programként tesztelhetők. (Már amennyire t.i. az tesztelhető, hogy egy szekvenciális program az "összes lehetséges bemenetre" véges időn belül az előirt választ adja.)

3.4 Korlátozott ujraindithatóság

Minden taszkról feltételezzük, hogy az általa ellátandó tevékenységre általában elegendő idő áll rendelkezésére. Mindamellett nem tekintjük rendellenesnek, ha egy taszk müködési feltétele már a taszk-akció végrehajtása közben ujból (akár többször is) teljesül. Ilyen "ujraindulási igény" esetén a taszk soronkövetkező akcióját várakozás nélkül megkezdi, de csak egy akciót hajt végre, függetlenül attól, hogy előző akciója alatt hány ujraindulási igény érkezett be.

A taszkoknak ez a korlátozott ujraindithatósága valójában nem a taszk strukturájából fakad (v.ö. 4.2.3. d/ pontjával), ezért a korlát szükség esetén a struktura megváltoztatása nélkül növelhető vagy akár praktikusan végtelenné is tehető.

3.5 "Egyenjoguság"

A kidolgozott taszkok egyrésze "technológiai", más részü "szolgáltató" ("rendszer") funkciókat lát el. A taszkok ezen két csoportja között a taszkon belül semmiféle megkülönböztetést (prioritás, master-slave mód, stb.) nem teszünk.

3.6 Taszk struktura

Az (általános) taszk fogalomnak megfelelő programstrukturát vázlatosan az alábbi SIMULA osztály formájában adhajtuk meg:

```
process class taszk;
virtual:
procedure interprete (paramteres);...
procedure act;
begin
        ...
        inner
repeat: wait for (entry);
        while not param box.empty do
        interprete (message from (param box));
        act;
        signalize (exit);
        goto repeat;
end class task;
```

Ezek szerint a taszkok passziv és aktiv állapotai egymást váltogatják. Passziv állapotba kerülhet a taszk müködési feltételének vizsgálatakor (wait for (cond)).

Aktiv állapota müködési feltételének teljesülésekor kezdődik és ennek során

- szükség esetén módositja paramétereit (while not param box. empty do ...);

- végrehajt egy reá jellemző, véges időn belül befejezendő akciót (act);
- jelzi környezete számára aktuális tevékenységének befejezését (signalize).

Az osztálytörzsben egy szabadon definiálható programrész (inner) és két helyettesithető eljárás (interprete, act) szerepel. A speciális taszk tipusok programozása
- az illető tipushoz tartozó saját változók deklarálásából,
- az inner helyén az egyes változók esetlegesen szükséges kezdőértékének megadásából,
- a taszk operátor által megadott paramétereinek feldolgozását meghatározó interprete eljárás megadásából,
- a taszk tulajdonképpeni (ismétlődő) tevékenységét meghatározó "act" eljárás megadásából áll.

A taszkok korrektségét biztositó programozási konvenciók - amelyek természetesen a "szabadon definiálható" programrészekre is vonatkoznak - az alábbiak:

a/ Minden taszk a saját maga által definiált mennyiségeken és eljárásokon kivül csak globális mennyiségekre hivatkozhat, másik taszkra nem.

b/ A taszkok környezetükkel csak szabványos eljárásokon keresztül érintkezhetnek. Szabványos eljárás szolgál
   - a müködési feltételek kivárására;
   - a taszkok és környezetük közötti adatforgalom lebonyolitására;
   - a szabványos eljárásokon belül vagy esetleg azokon kivül is szükséges erőforrás kezelésre;
   - a hiba kezelésre;
   - az aktuális tevékenység befejezésének jelzésére.

c/ Minden taszknak biztositania kell, hogy aktuális müködése véges időn belül végetérjen. Ez elsősorban az erőforrás használatra vonatkozóan jelent további, később részletezendő megszoritást, de jelenti azt is, hogy pl. az "act" eljáráson belül ujabb müködési feltétel teljesülésére már nem szabad várni, minthogy ez nem "garantáltan véges idejü" eljárás.

d/ Minden taszknak biztositania kell, hogy aktuális müködésének befejeztével valamennyi, ezen müködés során lefoglalt erőforrást felszabaditsa.

A fenti konvenciók biztositják egyrészt a taszkok függetlenségét, másrészt biztositják, hogy ezen független elemekből egy összetett rendszer felépithető legyen.

4. A TASZKOKKAL KAPCSOLATOS TAPASZTALATOK

4.1 Több taszk-alaptipus megkülönböztetése

A fentiekben ismertetett taszk fogalom a gyakorlatban fellépő feladatok számára általában megfelelt. Néhány esetben azonban célszerünek látszott olyan taszkok kialakitása, amelyek - működésük eredményétől függően - különböző események megtörténtét jelezhetik környezetük számára. Például az a taszk, amely egy berendezés állandósult állapotát ellenőrzi ilymódon teszi lehetővé, hogy a rendszer más tevékenységeket hajtson végre, ha az adott berendezés nincs állandósult állapotban, mint egyébként. Ilyen taszk nyilván nem hozható létre a megadott keretek között (t.i. a szabadon definiálható részek alkalmas megválasztásával).
A 3.6 fejezetben vázolt "univerzális" taszk-struktura helyett ezért több taszk alaptipust kellett definiálni.
A szükséges alaptipusok definiciója pl. SIMULA nyelven az alábbi osztálydeklarációk formájában adható meg:

```
process class cyclic process;
virtual:
procedure interprete (parameters);...
procedure act;
label repeat;
begin
      ...
repeat:
      inner
      goto repeat;
end;
```

```
cyclic process class simple task;
begin
        ...
        inner
repeat: wait for (entry);
        while not param box empty do
        interprete (message from (param box));
        act;
        signalize (exit);
end;
```

Az igy megadott "cyclic process" a ciklikusságot és a paraméterezhetőséget deklarálja. A "simple task" és a 3.6 alfejezetben leirt "taszk" azonos. Emellett viszont lehetőség van más taszk-tipus létrehozására is:

```
cyclic process class alternative task;
begin
        boolean aye;
        ...
        inner
repeat: wait for (entry);
        while not param box.empty do ...
        act;
        if aye then signalize (yes)
        else signalize (no);
end;
```

Ezen alaptipusokhoz - mint azt majd a következő fejezetben látjuk - még egy további tipus járul. Ezzel együtt minden, a Péti Nitrogénmüvek számára kidolgozott taszk besorolható valamelyik alaptipusba és a taszkok vezérlési kapcsolatainak számát és jellegét ezen alaptipus meghatározza.

4.2 A taszkok programozása

A taszkok programozását rendszerprogramozói feladatanak tekintettük, elsősorban biztonsági okokból. Egyrészt a rendelkezésünkre álló operációs rendszer nem tette lehetővé, hogy az egyes taszkokhoz tartozó területeket illetéktelen hozzáféréssel szemben védeni tudjuk. Másrészt a taszkok programozása - más praktikusan szóbajöhető lehetőség hijján - assembler nyelven folyt, ezért a rendszer teljesen védtelen volt a programfejlesztés során elkövetett hibákkal szemben.

Ennek ellenére a későbbi, felhasználási fejlesztések érdekében a felhasználót bevontuk a taszkok kidolgozási munkáiba. A vázolt taszk struktura lehetővé tette az ilyen együttmüködést anélkül, hogy "bedolgozóktól" a szervezés részleteinek megismerését megkivánta volna.

A jelen adottságok mellett a felhasználói fejlesztés kérdése tisztázatlan, mert egyrészt kockázatos, másrészt valószinüleg nélkülözhetetlen.

Megfelelő ellenőrzést biztositó operációs rendszer és magasszintü nyelv birtokában természetesen veszélytelenül megengedhető volna, a felhasználó számára, hogy az - adott taszk fogalom keretein belül - ujabb taszktipusokat hozzon létre.

4.3 A taszkok tényleges függetlensége

A taszkokra vonatkozó programozási konvenciók betartását semmi sem biztositotta, ezért az egyes taszkokban elkövetett hibák hatása olykor tulgyürüzött az adott taszk határain és más, esetleg már gondosan letesztelt taszkok müködésében okozott zavart. Ezek a hibák elkerülhetők lettek volna a szabványos eljárások olyan variánsainak kidolgozásával, amelyek a konvenciók betartását ellenőrzik.

A taszkok függetlenségét biztositani hivatott programozási konvenciók sajnos még betartásuk esetén sem biztositottak teljes függetlenséget annak következtében, hogy az adott operációs rendszer mellett egy overlay strukturáju program másképpen fut le, ha futása alatt swapping-re kerül, mintha sem. (Egyes belső változók ujból kiinduló értéküket veszik fel). Egy taszk-akció eredménye ilymódon függhet attól, hogy vele párhuzamosan hány és milyen méretü más taszk fut. Megfelelő programozási technikával ez a függőség megelőzhető, de egy belövés alatt álló program esetében ez is a lehetséges hibák egyik, környezetfüggő forrása.

4.4 A taszkok dinamikus helyfoglalása

A Péti Nitrogénmüveknél üzembeállitott rendszer esetében az operációs rendszer lehetővé tette, hogy a taszkok müködésük során dinamikusan memóriaterületet foglaljanak le. Utólag visszatekintve, ezen lehetőség kihasználása nem bizonyult tulságosan szerencsésnek. Kétségtelen ugyan, hogy ilymódon hatékonyabb tárgazdálkodást lehetett kialakitani, azonban az operációs rendszer esetünkben nem nyujtott védelmet a taszkok rendelkezésére bocsátott terület illetéktelen felhasználásával szemben (csak a terület foglaltságát tartotta nyilván, a hozzáférésre, felszabaditásra (!) illetékes taszkot nem). Ez néhány alkalommal rendkivül nehezen felderithető, környezettől és szituációtól függő hibát eredményezett, amely ritkán fordult ugyan elő, de akkor katasztrofális hatásu volt.

4.5 Konkluzió

A taszkokkal kapcsolatos valamennyi negativ tapasztalatunk tulajdonképpen azzal hozható kapcsolatba, hogy a fejlesztési munka jelentős részben nem fejlesztési célokat szolgáló operációs rendszer alatt történt.

(Más kérdés, hogy ilyennek hiányában az adott feladat elbirta volna-e valamiféle fejlesztő rendszer "terven felüli" kidolgozását, amely pl. a taszkokat egyedi tesztelésük során illegális akciók, különféle programozási konvenciók szempontjából ellenőrzi.) Mindent tekintetbe véve az a véleményünk, hogy a 3. fejezeteben megadott taszk struktura az itt felsorolt kiegészitésekkel lehetővé teszi egy nagyobb feladatnak kisebb, független részekre történő tagolását, és - megfelelő eszközökkel végzett munka esetében - ezeknek független kidolgozását, tesztelését.

5. A TASZKOK SZINKRONIZÁCIÓJA

A párhuzamos programok szinkronizációja egyes oszthatatlan egységek előirt időbeli sorrendjének biztositását jelenti. A szinkronizációnak ennek megfelelően különböző szintjeit lehet megkülonböztetni attól függően, hogy a program milyen részeit tekintjük oszthatatlannak.

Operációs rendszer szinten a gép egy-egy elemi utasitása (esetleg mikro-utasitása) számit oszthatatlan egységnek.* Rendszer programozói szinten azon kisebb-nagyobb programszakaszok az oszthatatlanok, amelyek nem tartalmaznak szinkronizációs utasitást (wait, activate, request, release, stb.).
Felhasználói, operátori szinten egy-egy önálló program (taszk) tekintendő oszthatatlan egységnek.

5.1 Programozói szintü szinkronizáció

A taszkok egy-egy müködése során programozói szintü szinkronizációra az alábbi szituációkban kerül sor:
- hozzáférés közös elérésü memóriaterülethez, (müködési feltételek teljesitése, ill. teljesülésének vizsgálata; kommunikáció taszkok között, stb.),
- hardware erőforrások használata (sornyomtató, operátori irógép stb.).

A szinkronizációt valamennyi esetben erőforráskezeléssel biztositottunk. (Az operációs rendszer lehetővé tette "absztrakt" erőforrások definiálását, valamint a taszkok számára ezek lefoglalását, várakozást felszabadulásukra, stb.).

Az erőforrás-kezelés általában nem közvetlenül a taszkokban, hanem azokban a szabványos eljárásokban történik, amelyek a szinkronizációt igénylő tevékenységek ellátására a taszkok rendelkezésére állnak (müködés befejezését jelző eljárás,

---

*Operációs rendszer szintü szinkronizációval csak kényszerből foglalkoztunk, erre itt nem térünk ki.

kommunikációs eljárás, stb.). Erőforrások "közvetlen" kezelésére csak néhány kivételes esetben volt szükség.

A taszkok tevékenysége során más, explicit vagy implicit szinkronizációs utasitást (mint pl. egy másik taszk müködésének felfüggesztése, várakozás aktivitásra, stb.) nem engedtünk meg.

Ilyen formán az a taszkokkal kapcsolatos követelmény, hogy egy-egy müködésük véges időn belül végetérjen - triviális programozási hibákat leszámitva - az erőforrás használat holtpont-mentességének követelményével egyenértékü.

A holtpont-mentesség biztositására gyakran használatos azon ökölszabály, hogy erőforrás birtokában ujabb erőforrás lefoglalása nem megengedett. Ez azonban tulságosan erős programozási megszoritást jelent és sok esetben nagyon gazdaságtalan megoldásra vezet. (Közlemények lemásolása, felesleges puffer foglalás, stb.) Helyette sikeresen alkalmaztuk az erőforrások hierarchikus lefoglalási szabályát, ami azt jelenti, hogy erőforrásaink között hierarchia szinteket definiálunk, s egy-egy taszkon belül az erőforrások lefoglalását csak növekvő, felszabaditását csak csökkenő szintek szerint végezzük. Könnyen belátható, hogy ilyen megszoritás mellett az erőforráskezelés nem vezethet holtpontra. Könnyen látható továbbá, hogy az erőforrások hierarchikus lefoglalásának szabálya - betartása esetén - egyben azt is biztositja, hogy az egyes taszk akciók befejeztével ne maradjon erőforrás a taszk birtokában.

5.2 Felhasználói szinkronizáció

A taszkok elinditására, időzitésére irányuló operátori tevékenységet általában nem szokás szinkronizációnak tekinteni, jóllehet ez esetben is annak meghatározásáról van szó, hogy mikor, milyen feltétel mellett indulhat a szóbanforgó program.

A különböző rendszerekben az operátornak e célra rendelkezésre álló eszközök elég szegényesek, s igy olyan szinkronizációs kérdések is gyakran programszinten kerülnek megoldásra, amelyek tulajdonképpen pillanatnyi felhasználói igénytől függenek ill. jó volna, ha attól függhetnének. (Például a PCM-ben adott eszközökkel operátori szinten nem lehet azt biztositani, hogy egy program müködésének végeztével automatikusan elinditson egy másikat.)

A taszk-tevékenységek időbeli sorrendjét a taszkok müködési feltételeinek, "együttmüködési szabályának" megadásával lehet előirni. A rendszerben zajló tevékenységek ilyen értelmü programozását nem tekintettük rendszerprogramozói feladatnak.

A taszkok együttmüködési szabályainak előirását a pillanatnyi igényeknek megfelelő taszk együttes összeállitásával együtt a felhasználóra biztuk és ezek megadására megfelelő eszközt bocsátottunk rendelkezésére.

A továbbiakban az e célra választott eszközt ismertetjük és megadunk néhány azzal többé-kevésbé egyenértékü alternativát is a választás jobb megvilágitásának céljából.

Az összehasonlitás megkönnyitésére tekintsük a következő - nem egészen légből kapott - feladatot.

Legyen A egy szakaszosan müködő adatgyüjtő, a B egy szakaszosan müködő feldolgozó egység és legyen a két egység egymástól független abban az értelemben, hogy egyik se tartalmazzon megkötést a másik müködésére vonatkozóan. Tételezzük fel, hogy normális esetben mind A, mind B müködési ideje behatárolható, de rendellenes esetben bármelyik nagymértékben meghosszabodhat. Szeretnénk közöttük olyan kapcsolatot kialakitani, amelyben

- az A által gyüjtött adatokat B dolgozza fel,
- az A müködését B nem befolyásolja,

- B-nek lehetőség szerint minden A által szolgáltatott adatot fel kell dolgoznia,
- a felhasználó értesül arról, ha a B nem végez idejében a feldolgozással, s ezért a feldolgozás folyamatossága megszakad (A felhasználó értesitése legyen a C tevékenység).

5.2.1 Első alternativa: magasszintü nyelv

A megfogalmazott feladat tulajdonképpen három tevékenység (adatgyüjtés, feldolgozás, hibajelzés) "párhuzamos programozása". Csábitó lehetőségnek tünhetne erre a célra valamilyen magasszintü nyelv felhasználása. Pl. a megadott követelményeket kielégitő program egy konkurrens PASCAL-szerü leirásban az alábbi lehetne:

```
cobegin
    var S: shared record a, b: boolean end
    repeat
        A;
        region S do a:=true;
    forever;
    repeat
        region S do
        begin
            await a;
            a:=b:=false;
        end;
        B;
        region S do b:=true;
    forever;
    repeat
        region S do
        begin
            await a and not b;
            a:=false;
        end;
        C;
    forever;
coend;
```

Itt azon felül, hogy az A, B, C semmit nem tételez fel egymásról, még az együttmüködésüket megadó program sem tételez fel róluk semmit, tehát az együttmüködési szabálynak és az ellátandó tevékenységeknek a leirása teljesen elválik egymástól.

Ennek ellenére kérdéses, hogy célszerü volna-e a felhasználótól jártasságot megkivánni ilyen magasszintü nyelvben ahhoz, hogy rendszerét pillanatnyi igényeinek megfelelően alakitani tudja.

(Ilyen irányu csábitásnak egyébként nem voltuk kitéve, minthogy az adott gépen nemhogy magasszintü párhuzamos programozási nyelv, de még a taszkok szekvenciális programjainak leirására alkalmas magasszintü nyelv sem állott rendelkezésünkre.)

5.2.2 Második alternativa: fix időzités

Fix időzités mellett a kivánt együttmüködés csak ugy képzelhető el, ha a hibajelzést ellátó C tevékenység tesztelni tudja a B állapotát, tehát ha explicit hivatkozás történik benne egy másik, tőle független tevékenységre. Müködésének kimenetele ily módon környezetétől függ:

```
if B active then out message (...)
```

Ez esetben - valamiféle "normális" müködési időket tekintetbe véve - felhasználói szinten előirható számukra egy alkalmas menetrend. Például:

```
all 2 MIN activate A;
after 40 SEC all 2 MIN activate B;
after 100 SEC all 2 MIN activate C;
```

(A parancsok jelentése nem szorul részletesebb magyarázatra, formája egyébként a PEARL task-scheduling utasitásainak felel meg.)

A fix időzités felhasználása azon felül, hogy miatta meg kellene sérteni a taszkokra kimondott függetlenségi elvet, egyébként sem tünt kivánatosnak.

Mindenképpen fel kellett készülnünk véletlenszerüen (operátori beavatkozás, vagy üzemi szituáció következtében) fellépő igények kiszolgálására és fix időzités mellett véletlen hosszuságu várakozás előirására nincs eszköz. A probléma legfeljebb felesleges "üres" müködések beiktatásával kerülhető meg.

A fix időzités nem elég gazdaságos, mert az egyes taszkok müködési idejét jelentős mértékben befolyásolja a vele együtt müködésben lévő többi taszk. Márpedig a véletlenszerüen fellépő igények kiszolgálása miatt az egyidejüleg müködő taszk-együttes szinten véletlenszerü, s ezért egy fix időzitésben minden taszk számára tetemes "biztonsági tartalék" időt kellene megadni a ténylegesen felhasznált időhöz képest. (Egy ilyen időtartamnak a megbecslése egyébként is alighanem teljesen "idegen" feladat lenne egy technológus beállitottságu felhasználó számára.)

Végül pedig egy ilyen szervezés nem elég rugalmas, mivel egy uj taszk beállitása esetén az időviszonyok esetleg gyökeresen megváltoznak, tehát valamennyi taszk időzitése revideálásra szorul.

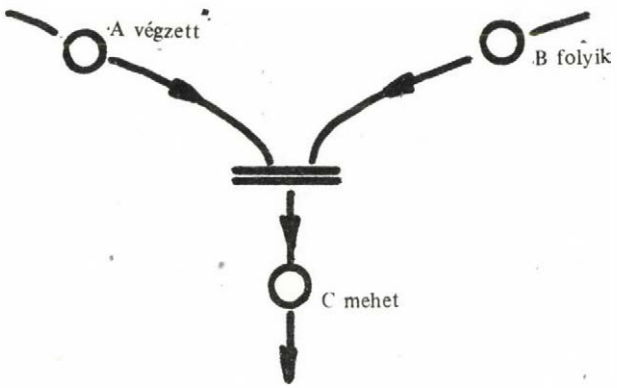
5.2.3 Harmadik alternativa: operációs rendszerben biztositott eszközök felhasználása

Az operációs rendszer eseménykezelése önmagában nem volt megfelelő. Egyrészt operátori szinten nem hozzáférhető, tehát nem alkalmas arra, hogy segitségével a felhasználó kész rendszerelemekből rendszert definiáljon. Másrészt abban a formában, ahogy rendelkezésünkre állt, leginkább csak fastrukturáju vezérlési gráf kialakitására alkalmas. (Csak egyszerü feltételek - egy-egy esemény - kivárását teszi lehetővé, összetett feltételekét nem.)

Mintapéldánk C tevékenységéhez ez pl. nem elegendő, mivel a közlemény kiadásának két feltétele van:
A befejeződött és B még nem ért véget.

5.2.4 Negyedik alternativa: Petri-háló

A taszkok együttmüködésének elirására a párhuzamos folyamatoknak un. Petri-hálóval történő leirását választottuk. A Petri-háló eredeti definiciója szerint egy irányitott gráf, amelyben a csomópontok helyén váltakozva "helyek", illetve "tranziensek" szerepelnek. A helyek feltételeket, a tranziensek valamilyen müködést reprezentálnak. A helyeket szokásosan körökkel, a tranzienseket duplán rajzolt egyenes szakaszokkal ábrázoljuk. Pl.:



1. ábra

A háló müködését a tranziensek "billenése" jelzi. Egy tranziens akkor billenőképes, ha valamennyi bemenő feltétele teljesül. A feltételek teljesülését a helyek "állapota" (az ábrán a megfelelő körbe rajzolt pont) jelzi. A tranziensek billenése pillanatszerü, és valamennyi bemenő feltételét megszünteti, valamennyi kimenő feltételét teljesiti. (A körökben lévő pontok számát eggyel csökkenti, illetve növeli.)

A megadott szabályok a háló müködését nem határozzák meg egyértelmüen, Mindazon müködések, amelyek ezen szabályoknak nem mondanak ellent, megengedettek.

Az eredeti definicióhoz képest az alábbi megszoritásokat tettük.

a/ A helyek és tranziensek mellett a hálóban belső hálók is megjelenhetnek. A háló határait ugy definiáljuk, hogy bemenő élei mindig helyekhez kapcsolódjanak, kimenő élei tranziensekhez. A taszkok a hálóban belső háló formájában szerepelnek.

b/ A Petri-háló müködésére vonatkozó eredeti előirás nyitva hagyja azt a kérdést, hogy egy billenőképes állapotban lévő tranziens mikor billen.
Esetünkben, ha a háló állapotában valamilyen változás történik, ezt követően valamennyi billenőképes tranziens "azonnal" müködésbe jön. A billenések eredménye ugyszintén változást jelent, tehát az igy billenőképessé váló tranziensek ugyszintén azonnal müködésbe jönnek, és ez mindaddig folytatódik, amig a hálóban még van billenőképes tranziens.

A továbbiakban csak olyan hálókkal foglalkozunk, amelynél ez a folyamat, azaz a háló aktuális "müködése" végleges lépésben véget ér.

c/ A Petri-háló definiciója szerint a tranziensek billenése pillanatszerü, tehát a háló soha sincs átmeneti állapotban. Esetünkben ezt olyan értelemben biztositottuk, hogy a billenések időben egymást kizárják, tehát az alatt az idő alatt, amig egy tranziens átbillen, a hálóhoz tartozó helyek állapotában más okból nem következik be változás. (Billenés csak "nem átmeneti" állapotban kezdődhet el.)

Belső hálót is tartalmazó háló esetében a tranziensek billenésének kölcsönös kizárása csak azokra a tranziensekre terjed ki, amelyek a külső háló helyeihez kapcsolódnak, a belső helyekhez kapcsolódókra nem. A belső hálók müködése ezért már nem tekinthető pillanatszerünek a fentebbi értelemben, hanem a müködés a külső háló bármely állapotában megkezdődhet, s a két háló müködése párhuzamosan folyhat.

(A határok meghuzásának következtében a belső háló müködésének megkezdése a külső helyeinek állapotát nem érinti.)

d/ Csak olyan hálókra szoritkoztunk, amelynél az egy-egy helyen lévő pontok száma maximálisan egy lehet. A pontok számának összeadását logikai összeadásnak (or) tekintettük.

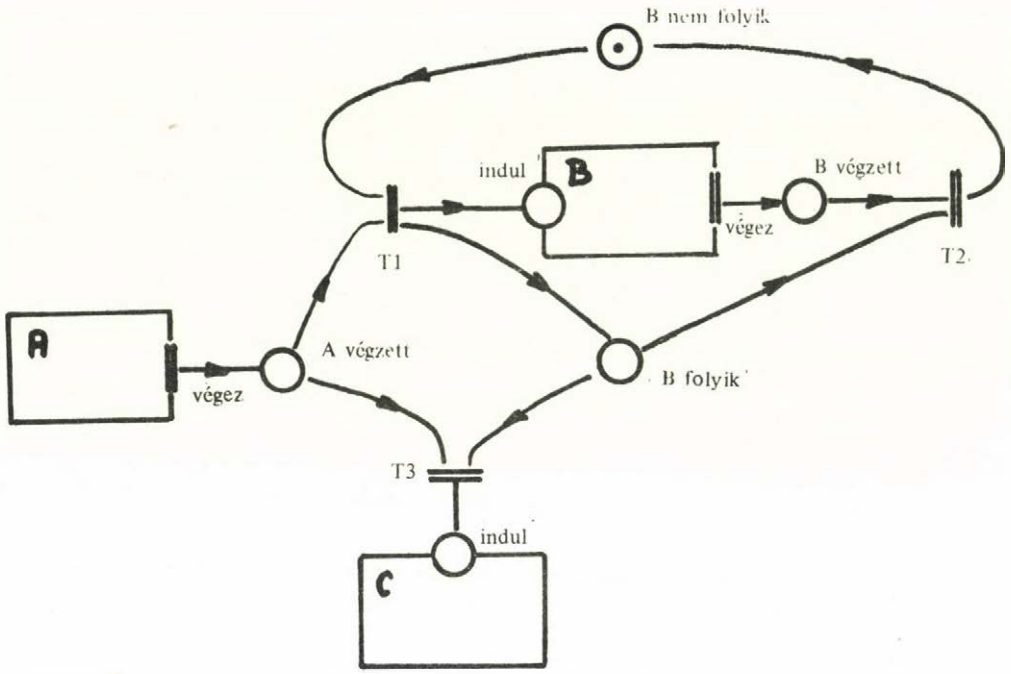
Az a/ - c/ megszoritás a háló eredeti definiciója által megengedett müködések közül egyet határoz meg. Pontosabban annak az ekvivalens Petri-hálónak egy megengedett müködését határozza meg, amit a taszkoknak a megfelelő hálóval történő helyettesitésével nyerünk. Ennek a Petri-hálókkal kapcsolatos elméleti eredmények alkalmazhatósága szempontjából van jelentősége. (Ha pl. az ekvivalens hálóról kimutatható, hogy minden megengedett müködés mellett holtpont mentes, akkor az eredet háló müködése az a/ - c/ megszoritások mellett is az.)

A d/ megszoritás azt jelenti, hogy pl. két egymás után kapcsolt taszk

2. ábra

esetében nem tekintettük eleve rendellenesnek, ha két "A" müködést csak egy "B" követ. Ha pl. "A" valamilyen üzemi jellemző mérését végzi, "B" pedig a jellemző kijelzett aktuális értékének megujitását, akkor a "B" funkció havária, vagy egyéb okokból fellépő pillanatnyi tulterhelés hatására időlegesen kimaradhat, későbbi pótlólagos elvégzése pedig értelmetlen. Ha valahol viszont ez nem engedhető meg, akkor azt a Petri-háló megfelelő kialakitásával (un. biztonságos háló) tudjuk biztositani. (Ez a megszoritás egyébként inkább technikai, mintsem elvi jellegü. Ilymódon az egyes helyek állapotát egy bittel tudtuk reprezentálni, és a tranziensek billenőképességének vizsgálatánál jól ki lehetett használni a párhuzamos bitmüveleteket.)

A mintapéldánkban megkivánt együttmüködés Petri-háló segitségével az alábbi módon irható elő:



3. ábra

Ezen hálóban a B taszk müködő vagy várakozó állapotát egy-egy hely ("B folyik", illetve "B nem folyik") állapota jelzi. Kezelésüket a B-t inditó T1, illetve a B végzésekor billentő T2 tranziens végzi. A feladat követelményei szerint C jelzi a felhasználónak, ha B nem végzett időben a feldolgozással. Ennek megfelelően itt A vagy a T1-et (B-t) vagy a T3-at (C-t) inditja aszerint, hogy "B nem folyik" vagy "B folyik".
(Az adott háló természetesen hallgatólag feltételezi, hogy a feldolgozás kimaradása nem fordul elő sürübben annál, mint amit C egyáltalában jelezni képes. Ez esetben ui. lesznek nem jelzett kimaradások.)

Az A, B, C taszknak az ábrán megadott kapcsolatát a felhasználó alkalmas parancsok segitségével adhatja meg:

```
A.végez.következménye (A végzett);
T1.billen ha (A végzett).és (B folyik);
T1.következménye (B indul).és (B folyik);
B.végez.következménye (B végzett);
T2.billen ha (B folyik).és (B végzett);
T2.következménye (B nem folyik);
T3.billen ha (A végzett).és (B folyik);
T3.következménye (C.indul);
```

A parancsok itt közölt formája a Bevezetésben emlitett szimulációs programnak felel meg. A Péten realizált rendszer ettől formai szempontból különbözik, tartalmilag vele azonos.

A hálók és a taszkok fogalma tulajdonképpen ekvivalens. Ekvivalens abban az értelemben, hogy egyrészt minden taszkhoz tartoznak helyek és tranziensek. (A 4. fejezetben vázolt "simples task" tipushoz pl. a taszk müködési feltételét jelképező egyetlen hely és az aktuális taszk-akció befejezésekor billenő egyetlen tranziens.)

Másrészt minden hálóhoz tartozik egy taszk, amelyik t.i. a háló (jelen fejezet b/ pontjában leirt) "müködését" gépen belül realizálja. A "nem triviális" hálókhoz tartozó taszkok külön alaptipust képviselnek. A hálók a/ - d/ megszoritásoknak megfelelő müködése ezen belül került lerögzitésre a taszk "act" eljárásának alkalmas megadásával.

Az egész müködő rendszer maga is egy háló, amelynek elemei közül a helyek és a tranziensek a rendszer generálásakor jönnek létre rendszerprogramozási szinten meghatározott számban. A technológiai taszk példányok létrehozása, valamint ezek közötti vezérlési kapcsolatok kialakitása felhasználói tevékenység. (A taszkoknak - mint már emlitettük rendszerprogramozói szinten csak prototipusát hozzuk létre.)

A taszkok együttmüködésének Petri-hálóval történő leirása több szempontból kedvező:

- jól áttekinthető grafikus ábrázoláshoz kapcsolható;
- segitségével a felhasználó nem az egyes tevékenységek időbeli sorrendjét, párhuzamos végrehajtását irja elő; a tevékenységek müködési feltételeit nem időzitésük alkalmas megválasztásával biztositja, hanem bizonyos mértékig magukat a müködési feltételeket adhatja meg;
- a Petri-háló formájában megadott feltételrendszer közvetlenül felhasználható taszkok müködésének tényleges vezérlésére;
- ily módon lehetőség nyilik a Petri-hálókra vonatkozó elméleti eredmények (életképesség, holtpont mentesség stb. vizsgálata) felhasználására.

A Petri-hálót csak a taszkok felhasználói szintü szikronizációjára használtuk fel. Elvileg elképzelhető lenne a taszkok programozási szintü, vagy akár operációs rendszer szintü együttmüködését is ilyen uton biztositani.
Ehhez azonban alighanem speciális "primitvák"-ra, illetve gépi utasitásokra volna szükség. További részletes elemzést igényelne, hogy ennek az elvnek ilyen univerzális felhasználása milyen előnyökkel és hátrányokkal járna.

A kérdés mindenképpen messzire vezet és az adott esetben az volt a célunk, hogy ha csak lehet, a meglévő eszközöket használjuk fel.

Ezzel kapcsolatban tapasztalataink nincsenek, mert a fejlesztés - különböző okokból kifolyólag - az üzembe kerülő gép-példányon és konfiguráción folyt, ezen pedig sem dokumentációt karbantartó rendszer nem volt, sem pedig mindenki számára elegendő hozzáférést nem biztositott.

8.7 A folyamatos üzemelés feltételei

A kidolgozandó rendszerrel szemben támasztott alapvető követelmény volt a folyamatos 24-órás üzem. Ilyen igényre szemmelláthatóan sem a gyári alapsoftware, sem a hardware nem volt felkészülve.

Az operációs rendszer pl. holtpontra juthat annak következtében, hogy a standard I/O utasitásokban nem vizsgálja (vagy nem következetesen vizsgálja) az egyes perifériák készenléti állapotát. Különösen addig fordult ez gyakran elő, amig a diszk és mágnesszalag egyidejü használatát (amit egyébként a gép specifikációja megenged!) software uton ki nem zártuk.

A mátrixprinter folyamatos készenléte csak állandó bekapcsolt állapot mellett biztositható, ez viszont a berendezést erősen igénybe veszi.

A standard I/O utasitások nem teszik lehetővé, hogy az operátori irógépre kiadott olvasási parancsokhoz időkorlátozást (time-out) adjunk meg, s igy egy operátori válasz kimaradása praktikusan fennakadást eredményez.

A legjelentősebb hiányosságot ezen a területen az jelentette, hogy a gép központi egységének és perifériáinak ellenőrzésére semmiféle, on-line üzemben felhasználható tesztprogram nem létezett. A gyári tesztprogramok csak a folyamatos üzem leállitásával, a teljes diszk tartalmának mentésével, visszamentésével és a rendszer ujrainditásával használhatók. Az akció mindenképpen információvesztéssel jár, mert a futó programok tetszőleges

pillanatban történő megszakitása és visszaállitása - állitólag - elvileg sem megoldható.

A diszk folyamatos tesztelésére saját fejlesztésü eszközt adtunk, a többi periféria esetében a karbantartással, tesztelés-sel járó kiesést a felhasználó kénytelen elviselni.

Annak következtében, hogy ott csak egy hierarchiaszint van az ilyen taszkok egymást vagy csak valamennyi hely állapotának kiértékelése árán tudják müködésbe hozni, vagy a háló bizonyos helyeit megkülönböztetett módon kell kezelni. (A Péten müködő rendszerben ezen utóbbi lehetőséget választottuk.)

A hálónak több hierarchiaszintre tagolása ezt a problémát automatikusan megoldaná, minthogy az egyes taszkok bemenő feltételeit reprezentáló hely a taszkhoz tartozó (belső) háló eleme, s nem a külső hálóé.

Annak ellenére, hogy a Péten felmerült feladatok tehát nem tették szükségessé a hierarchikus szervezés megvalósitását, a benne rejlő előnyök és lehetőségek alapján ugy gondoljuk, hogy egy az általánosságra valamennyire is igényt tartó megoldás ezt a lehetőséget nem nélkülözheti.

Rendszerünket, annak esetleges ujabb felhasználása esetén ezzel mindenképpen kiegészitendőnek tartjuk.

6.3 Időben változó vezérlési kapcsolatok

A Péten átadott rendszerben nem tettük lehetővé a taszkok vezérlési kapcsolatainak menet közben történő módositását. Ez szervezési szempontból nem jelentene nehézséget, de meg kellene vizsgálni, hogy véletlenszerü időpontban végrehajtott módositás milyen következményekkel jár pl. a háló holtpontmentességre, ill. milyen feltételek mellett hajtható végre annak veszélyeztetése nélkül.

7. A TASZKOK KÖZÖTTI KOMMUNIKÁCIÓ

A taszkok között Petri-háló formájában csak vezérlési kapcsolatokat adunk meg. A vezérlés átadásával párhuzamosan más információ nem kerül átadásra.

Függetlenségük érdekében - mint már emlitettük - a taszkok nem tartalmaznak más taszkra vonatkozó hivatkozást. Ily módon a taszkok között "közvetlen" kommunikációra nincs lehetőség. A kommunikáció a taszkok környezetéhez tartozó, közös elérésü memóriaterületeken keresztül bonyolódik le.

Az adatforgalom lebonyolitására négy eszköz áll a taszkok rendelkezésére:

- first in - first out feldolgozásu információs listák tetszőleges tartalmu információk memórián keresztül történő átadására;
- diszk file-ok nagyobb mennyiségü információ átadására;
- egy közös adatterület, amely az időben változó folyamatjellemzők mindenkori aktuális értékét tartalmazza;
- egy archivum, amely kiválasztott folyamatjellemzők multbeli értékeinek elérését teszi lehetővé.

Ezek közül a közös adatterület és az információs listák kezelése érdemel emlitést, a másik kettő felhasználása és kezelése hagyományosnak mondható.

a/ A közös adatterületen tárolt jellemzők "folyamatos" felujitása és azok felhasználása egymástól függetlenül folyik. Egy-egy adat kiolvasása bármikor megtörténhet. A több elemből álló adatrendszerek kiolvasását, beirását lehetővé tevő eljárásoknak az adatok időbeli összetartozását biztositani kell. Ezt az irás és olvasás kölcsönös kizárásával értük el. A különböző taszkok által végzett egyidejü irási

(olvasási) tevékenységeket megengedtük, mivel feltételeztük, hogy minden jellemző felujitását csak egy taszk végzi (másszóval azt, hogy a taszkok kimenő adatrendszerei diszjunktak). A közös adatterületen tárolt adatok kiolvasását és beirását - az e célra szolgáló szabványos eljárásokon belül - szemaforkezeléssel oldottuk meg.*

Az információs listákon keresztül a taszkok tetszőleges tartalmu és bizonyos keretek között tetszőleges méretü üzeneteket adhatnak át egymásnak. A Péti Nitrogénmüveknél realizált rendszerben ilyen listákat két célra használtunk.

Egyrészt információs listákon keresztül történik a taszkok operátori paraméterezése, másrészt bizonyos rendszerszolgáltatásokat ellátó taszkok (közlemény kiadás, adott sorszámu üzemnapló összeállitása, stb.) információs listán keresztül kapják meg feladataikat. Ezek a "szolgáltató" taszkok a felhasználó által megadott Petri-hálóban nem szerepelnek. Rájuk semmiféle más program nem vár. Müködési feltételüket a szolgáltatást kérő taszk teljesiti az információ átadására szolgáló szabványos eljáráson keresztül. A szolgáltatást kérő taszk számára az eljárás meghivása erőforrásfoglalás és várakozás szempontjából közömbös.

---

* Ez látszólag ellentmond annak a korábbi állitásunknak, hogy a taszkok müködése során szükséges szinkronizációra kizárólag erőforrás kezelést alkalmaztunk. Az ellentmondás feloldható azáltal, ha a közös adatmezőhöz történő hozzáférés jogát olyan erőforrásnak tekintjük, amit nem egy-egy taszk foglal le, hanem az "összefonódó", irási (olvasási) tevékenységet végző taszkok együttese és amit ez az együttes mindaddig lefoglalva tart, amig van folyamatban lévő irási (olvasási) tevékenység. Ez az erőforráskezelés önmagában nem nyujt biztositékot arra, hogy az erőforrás foglaltsága - s ezzel együtt az esetleg reá váró taszk müködése - véges időn belül végetérjen.

Végtelen sok "összefonódó" irási (olvasási) igény fellépése rendkivül kis valószinüségü; esetünkben ki is volt zárva. Kizárta a közös adatterületet használó taszkoknak az általuk ellátott funkció természetéből fakadó együttmüködési szabálya, amely biztositotta, hogy az egyes taszkok ne ismételjék meg tevékenységüket változatlan bemenet mellett, másszóval amely biztositotta, hogy a közös adatterület adatainak két kiolvasása (beirása) között beirásra (kiolvasásra) is sor kerüljön.

8. A KIDOLGOZÁSI MUNKA ÉS A PRÓBAÜZEM SORÁN SZERZETT TAPASZTALATOK

Az elvégzendő munkával szemben támasztott egyik követelmény volt, hogy a termék legyen a Videoton OS-10 rendszerével kompatibilis. A felhasználó ehhez többek között azért is ragaszkodott, hogy háttérprogramozáshoz FORTRAN felhasználási lehetősége legyen. Ezen követelménynek egyik - számunkra legsulyosabb - következménye volt az, hogy a fejlesztési munkát megfelelő fejlesztő rendszer nélkül kellett elvégezni.

A rendelkezésünkre álló "gyári" alapsoftware, beleértve a Process Control Monitor-t is nem programfejlesztési célra készült. Az intézetben kidolgozott IDOS-fejlesztő rendszer pedig egyrészt nem volt OS-10 kompatibilis, másrészt nem multi-taszk rendszerek fejlesztésére szolgál, ezért csak korlátozott mértékben tudtuk felhasználni.

Munkánk egyik legfontosabb tanulsága, hogy hasonló volumenü fejlesztési munkát megfelelő fejlesztő rendszer nélkül nem szabad elkezdeni, mert igy a munka rendkivül terhessé válik (rengeteg mindent kell fejben tartani, nem elfelejteni, a tulajdonképpeni feladattal semmi kapcsolatban nem álló problémákkal foglalkozni, kényszermegoldásokat kiötleni, stb.), elhuzódik és a rossz munkakörülmények hatása a termék minőségén is meglátszik.

Hogy milyen fejlesztő rendszer a "megfelelő", annak egy valamennyire is teljes megfogalmazása tulmenne a jelen beszámoló keretein. A következőkben mindössze azokat a problémákat foglaljuk röviden össze, amelyek a legtöbb gondot okozták, s amelyek megoldását egy "megfelelő" fejlesztő rendszertől el lehet várni.

8.1 Könyvtárkezelés

A rendszer kidolgozásának során az egyes rendszerelemek számos példánya jön létre. Egyrészt minden elem számos megfogalmazási-, tesztelési-, használatbavételi-, ujrafogalmazási perióduson megy keresztül s az ennek megfelelő, időben egymást követő példányai az elem fejlődését tükrözik, egymástól többé-kevésbé különböznek. Másrészt valamely rendszerelem egy adott állapotában is számos példányban van jelen, amelyek egymástól elvileg csak tárolási helyben, formátumban különböznek. Igy létezik pl.:

- forrásnyelvü törzspéldány (ill. rendszerint törzspéldányok lyukszalagon, diszken és mágnesszalagon is);
- forrásnyelvü másolatok más egységekben;
- "relocatable binary" formátumu példány az un. rendszermodulok, ill. felhasználói modulok könyvtárában a szerkesztő program számára.
- ténylegesen végrehajtásra kerülő példányok az operativ memóriában, illetve a diszk "swapping" zónájában.
- "relocatable memory image" formátumu példány (példányok) az u.n. futtatható programok könyvtárában;
- ténylegesen végrehajtásra kerülő példányok az operativ memóriában, illetve a diszk "swapping" zónájában.

Munkánk során rendkivül hiányzott valami olyan eszköz, ami az egyes egységek különböző példányainak kompatibilitását mindenkor automatikusan biztositja, vagy amivel ez a kompatibilitás legalább ellenőrizhető.

Azon eszközök számára u.i., amivel a módositásokat végrehajtottuk, nem egy-egy program vagy eljárás jelentett egy logikai egységet, nem arra szolgáltak, hogy egy megadott egység valamilyen legális állapotból annak egy uj, legális állapotát hozzák létre, hanem többnyire csak egyes memóriarekeszek izolált módositására. (Egy fejlesztő rendszerben az ilyen eszköz alighanem kifejezetten káros, mert tul nagy kisértést jelent a "pillanatnyilag felesleges" tevékenysége negligálására.)

A rendelkezésünkre álló eszközökkel gyakorlatilag az sem volt eldönthető, hogy egy-egy egységnek itt-ott megtalált példánya mikori állapotokat tükröz; az, hogy egy-egy módositás már megtörtént-e rajta, vagy sem; hogy két különböző formátumu példány tartalmilag azonos-e vagy sem; hogy egy adott egységben a legutóbb használathoz képest történt-e szándékos vagy akár valamilyen hibából eredő (!) változás vagy sem.

Az egy-egy egység módositására szolgáló eszközökön tul nagyon hiányzott valami olyan eszköz, ami egy adott módositás esetén megadja mindazokat az egységeket, akiket ez érint. Ennek hiányában az egy módositások hatásának továbbgyürüzését csak az "el-nem-felejtés" és a nyomában fellépő ujabb hibák biztositják.

Ilyen körülmények között a könyvtárakban még akkor is törvényszerüen növekszik a "rendetlenség", ha azokat rendkivül fegyelmezett és szisztematikusan együttdolgozó kollegák használják.

(Az emlitetteket ugy is lehet fogalmazni, hogy hiányzott az egyes rendszerelemek kidolgozásának és módositásának párhuzamosan folyó tevékenységeit szinkronizáló, vagy legalább azt támogató gépi eszköz.)

8.2 Forrásnyelvi hibakeresés

A hibakeresést, hibajavitást nagymértékben megkönnyitette volna, ha végig forrásnyelvi szinten mehetett volna végbe.

Ehhez egyrészt az szükséges, hogy az operációs rendszer a különféle hibaszituációkat idejében felismerje (pl. ne csak akkor, amikor egy hibás ugró utasitás eredményeképpen valamelyik rendszertábla adatait nem sikerül utasitáskódként értelmezni), mert különben a kiadott hibajelek a hiba forrására vonatkozólag semmit sem mondanak.

Másrészt az is szükséges, hogy a kiadott hibajelzések a forrásnyelvi szöveg alapján értelmezhetők legyenek és a hiba felderitése érdekében végzett programozói akciók (a program megállitása valamilyen ponton, a programban deklarált változók értékének vizsgálata, bizonyos területek védelem alá helyezése, stb.) szintén forrásnyelvi szinten megfogalmazhatók legyenek. Esetünkben a hibakeresés megkövetelte a programok különböző formáinak, elhelyezkedésének a róluk vezetett nyilvántartásoknak, egyszóval az operációs rendszer részleteinek ismeretét. Enélkül többnyire csak kilátástalan találgatás lett volna.

8.3 Software-környezet egyedi tesztek céljára

Az egyes rendszerelemek szisztematikus egyedi tesztelésének céljára elmulasztottunk létrehozni egy megfelelő software-környezetet. Egy ilyen software környezet

- az egység számára biztosithatja mindazt a "szolgáltatást", ami majdan rendelkezésére fog állni;
- ellenőrizheti, hogy az egység a kötelező "játékszabályokat" nem sérti-e meg;
- operátori input-output lehetőséget biztosithat az egység számára a tesztelés lefolytatásához.

Utólag visszatekintve, egy ilyen környezet létrehozása alighanem már egyetlen felhasználás esetén is kifizetődött volna.

8.4 Párhuzamos tevékenységek nyomkövetése

Sem az operációs rendszer, sem az általunk kidolgozott rendszer nem tartalmazott megfelelő eszközt a párhuzamosan futó programtevékenységek utólagos nyomkövetésére. Az az előzetes elképzelés, hogy a taszkok által létrehozott software események egymásutánjának feljegyzése elegendő egy ilyen nyomkövetéshez, nem vált be. Ennél részletesebb információt, pl. az utolsó N végrehajtott utasitás folyamatos nyilvántartását már

csak egy speciálisan erre a célra készült fejlesztő rendszer keretében lehet biztositani.

8.5 Szimulált külső környezet

A komplett rendszer tesztelését mindenképpen kivánatos az üzembeállitást megelőzően elvégezni. Kérdéses viszont ezzel kapcsolatban az, hogy hol huzzuk meg a "komplett" rendszer határait. Beletartoznak-e folyamatperifériák vagy sem? Mivel a tesztelés célja a lehetséges hibaforrások szétválasztása, a próbaüzemelési tapasztalataink alapján feltétlenül indokoltnak tartjuk a rendszernek szimulált perifériákkal történő tesztelését.

Megfontolandó, hogy a perifériák szimulációja milyen eszközökkel történhet. Erre vonatkozólag nincs tapasztalatunk, mivel a tesztelésnek ez a fázisa esetünkben sajnos nem történt meg. Mindenesetre ezen tesztelés során kell ellenőrizni, hogy a perifériák minden megengedett és minden elképzelhető müködésére a rendszer "értelmesen" reagál-e vagy sem. (A próbaüzem idejére ezek után csak az "elképzelhetetlen" müködésekre adott reakciók kideritése marad, és ez sem kevés.)

8.6 Dokumentáció

Az egyes rendszerelemekről készülő különböző mélységü dokumentációnak a használata éppen a rendszer kidolgozása alatt a legintenzivebb. Tehát indokolt, hogy ezek a dokumentumok már ebben a periódusban rendelkezésre álljanak. Az egyes dokumentumok tartalma viszont éppen ezen időszakban változik a leggyakrabban, ami miatt a dokumentum (hagyományos eszközökkel történő) elkészitését mindenki csak valamiféle "végleges állapot" elérése utánra szeretné halasztani. Az ellentmondás feloldásának bevált módja a gépen belül tárolt és naprakész állapotban tartott dokumentáció - feltéve, hogy a munka során nem a fejlesztőgép jelenti a szük keresztmetszetet.

Ezzel kapcsolatban tapasztalataink nincsenek, mert a fejlesztés - különböző okokból kifolyólag - az üzembe kerülő gép-példányon és konfiguráción folyt, ezen pedig sem dokumentációt karbantartó rendszer nem volt, sem pedig mindenki számára elegendő hozzáférést nem biztositott.

8.7 A folyamatos üzemelés feltételei

A kidolgozandó rendszerrel szemben támasztott alapvető követelmény volt a folyamatos 24-órás üzem. Ilyen igényre szemmelláthatóan sem a gyári alapsoftware, sem a hardware nem volt felkészülve.

Az operációs rendszer pl. holtpontra juthat annak következtében, hogy a standard I/O utasitásokban nem vizsgálja (vagy nem következetesen vizsgálja) az egyes perifériák készenléti állapotát. Különösen addig fordult ez gyakran elő, amig a diszk és mágnesszalag egyidejü használatát (amit egyébként a gép specifikációja megenged!) software uton ki nem zártuk.

A mátrixprinter folyamatos készenléte csak állandó bekapcsolt állapot mellett biztositható, ez viszont a berendezést erősen igénybe veszi.

A standard I/O utasitások nem teszik lehetővé, hogy az operátori irógépre kiadott olvasási parancsokhoz időkorlátozást (time-out) adjunk meg, s igy egy operátori válasz kimaradása praktikusan fennakadást eredményez.

A legjelentősebb hiányosságot ezen a területen az jelentette, hogy a gép központi egységének és perifériáinak ellenőrzésére semmiféle, on-line üzemben felhasználható tesztprogram nem létezett. A gyári tesztprogramok csak a folyamatos üzem leállitásával, a teljes diszk tartalmának mentésével, visszamentésével és a rendszer ujrainditásával használhatók. Az akció mindenképpen információvesztéssel jár, mert a futó programok tetszőleges

pillanatban történő megszakitása és visszaállitása - állitólag - elvileg sem megoldható.

A diszk folyamatos tesztelésére saját fejlesztésü eszközt adtunk, a többi periféria esetében a karbantartással, tesztelésel járó kiesést a felhasználó kénytelen elviselni.

9. ÖSSZEFOGLALÁS

Mi az, ami a Péti Nitrogénmüvek részére kidolgozott munkából általánositható és ujra felhasználható?

a/ A szerzett tapasztalatok alapján az ott kialakitott alkalmazói rendszer strukturáját hasonló jellegü feladatok esetében általánosan felhasználhatónak tartjuk.

Ez a struktura jó a felhasználónak, mert lehetővé teszi olyan alkalmazói rendszer kidolgozását, ami

- testreszabott abban az értelemben, hogy lehetőség van a felhasználói fogalomkörnek megfelelő, "természetes" müveleti egységek kialakitására;
- rugalmas, mert egyrészt az adott müveleti egységek felhasználásával a felhasználó szabadon alakithatja ki rendszerét a pillanatnyi szükségleteinek megfelelően, másrészt a müveleti egységek köre könnyen bővithető;
- megbizható legalábbis annyira, amennyire ezt a felhasznált elemek tesztelése biztositja, és megbizható a rendszer holtpontmentessége szempontjából, feltéve, hogy a felhasznált operációs rendszer is az.

A kialakitott struktura jó a rendszer kidolgozójának, mert lehetővé teszi egyes előregyártott rendszerelemek ismételt felhasználását, valamint biztositja az egyes rendszerelemek függetlenségét és tesztelhetőségét.

b/ A kidolgozottak közül ujra felhasználható számos olyan rendszerelem, ami általános irányitástechnikai, illetve szervező funkciót lát el, s ily módon nem kötődik szigoruan az adott feladathoz. Ilyenek pl. az ACER programcsomag legnagyobb része vagy pl. a különféle üzemi naplók kiadására szolgáló taszkok. Ez esetben csak algoritmus szintü ujrafelhasználásról lehet szó, minthogy a realizált elemek - az R-10 assembler nyelv használata miatt - az adott géptipushoz kötöttek.

c/ Általánosan felhasználhatónak tartunk a rendszer továbbfejlesztésére vonatkozó néhány elképzelést, amelyek a szimulációs célokra készült SIMULA nyelvü programban vannak megfogalmazva. Ezek közül elsősorban a vezérlési kapcsolatok hierarchikus kialakitása érdelem említést. A szimuláció tanulsága szerint az adott megoldások müködőképesek, de természetesen még gyakorlati ellenőrzésre szorulnak.

Mi az ami hiányzik?

a/ Hiányzik valamilyen magasszintü nyelvi eszköz, ami

- lehetővé teszi az egyes rendszerelemek definiálását;
- lehetővé teszi a konkrét felhasználások esetén szükséges tárgyprogramok automatikus előállitását a rendszerelemek adott definiciója alapján;
- biztositja a kidolgozott elemek portabilitását.

A SIMULA nyelv elvileg alkalmas volna ilyen célra, hiszen a rendszer strukturájának és számos elemének SIMULA nyelvü definiciója már el is készült. Elvileg más (PASCAL, PEARL, ADA, stb.) magasszintü nyelv is szóbajöhet. Ennek alapján azonban a jelenlegi lehetőségeink mellett egy folyamatirányitási célu kisgépen vagy mikroprocesszoros rendszerekben felhasználható tárgyprogram csak ember segitségével "kompilálható".

Érdemes volna megvizsgálni, hogy egy megfelelő makro-nyelv és makro-készlet nem volna-e használható erre a célra.

b/ Hiányzik egy fejlesztő rendszer, ami azzal, hogy a rendszerfejlesztés szerteágazó munkáját megfelelően támogatja nem csupán a konkrét alkalmazói rendszerek előállitási idejét csökkentené igen jelentős mértékben, hanem a termék megbizhatóságát és a későbbi továbbfejlesztés lehetőségét is jelentősen megnövelné.

KÖSZÖNETNYILVÁNITÁS

A Péti Nitrogénmüveknél átadott rendszer szervezésének kialakitását dr.Almásy Gedeon és Huppert András kollégámmal együtt végeztük, a programozási munkákban rajtunk kivül részt vett még dr. Bányász Csilla, dr. Csillag Péter, Demjén Csaba, dr. Móritz Péter, valamint a Péti Nitrogénmüveknél dolgozó kollegák közül Feind Ferenc és Bastricz Ildikó.

Az Ő közremüködésük tette lehetővé, hogy az a rendszer, aminek tapasztalatairól itt beszámoltam, létrejöjjön.

Ezuttal mondok köszönetet dr. Almásy Gedeon kollégámnak azért, hogy ezen tanulmány kialakitása során vitapartnerként mindig készséggel rendelkezésemre állott.

FELHASZNÁLT IRODALOM

Brinch Hansen: Operating system principles.
Prentice-Hall Inc., England Cliffs, New Jersey, 1973.

G. Musstopf: Programming tools and standards for future dedicated computers and distributed systems.
Real-Time Programming 1978
/Proc. of the IFAC/IFIP Workshop/.

F. Prunet, J.M. Dumas and A. Reboul: Naturally describing synchronisation on real-time control.
Real-Time Programming 1978
/Proc. of the IFAC/IFIP Workshop/.

S. Lauensen: Debugging Techniques
Software-Practice and Experience, 9 1 /Jan. 1979/, 51-63.

S.J.Sreekaanth, T.A. Marsland: The Deadlock Problem
Computer 13, 9 /Sept. 1980/ pp. 58-78.

Asher Reuveni: The event based language and its multiples processor implenetation.
MIT/Laboratory for Comp. Sci., Cambridge /USA/, Doctoral thesis, 1980.