Buza Antal

# OPERÁCIÓS RENDSZEREK

Az egyszerű számítógép-modell
Az operációs rendszer felépítése, komponensei
A megszakításrendszer
Az operációs rendszer elindítása és leállítása
A processzorgazdálkodás módszerei
A memória dinamikus felhasználása
A párbeszédes és a kötegelt feldolgozás
Többprocesszoros rendszerek
Holtpontmentes vezérlés
Szuper operációs rendszerek
Virtuális megoldások

Dunakavics
D·U·E PRESS

INFORMATIKAI KÖNYVEK 6.

Buza Antal

# OPERÁCIÓS RENDSZEREK

© Dr. Buza Antal, 2016

**DUNAÚJVÁROSI EGYETEM**
**www.uniduna.hu**
**D·U·E PRESS**
Kiadóvezető Németh István

Felelős kiadó Dr. habilAndrás István
Felelős szerkesztő Nemeskéry Artúr
Tördelés Duma Attila

Készült a HTSART nyomdában
Felelős vezető Halász Iván

ISBN 978-963-9915-82-4

Buza Antal

# OPERÁCIÓS RENDSZEREK

Az egyszerű számítógép-modell
Az operációs rendszer felépítése, komponensei
A megszakításrendszer
Az operációs rendszer elindítása és leállítása
A processzorgazdálkodás módszerei
A memória dinamikus felhasználása
A párbeszédes és a kötegelt feldolgozás
Többprocesszoros rendszerek
Holtpontmentes vezérlés
Szuper operációs rendszerek
Virtuális megoldások

Dunakavics
D-U-E PRESS
Dunaújváros, 2016

INFORMATIKAI KÖNYVEK 6.

# *TARTALOM*

Előszó ..... 9

**1. Bevezetés, az operációs rendszerek kialakulása**
1.1. Az egyszerű számítógép-modell ..... 11
1.2. Kihasználatlanságok ..... 13
1.3. Az ötlet ..... 14
1.4. Megvalósítható-e? ..... 14
1.5. A megoldás ..... 16
1.6. A megoldás szülte újabb feladatok és megoldásaik ..... 16
1.7. Az operációs rendszerek feladatai ..... 19
1.8. A számítógép működése operációs rendszerrel ..... 19
1.9. Az operációs rendszer felépítése, komponensei ..... 20

**2. A megszakításrendszer**
2.1. A megszakítások forrása, okok szerinti osztályozása ..... 21
2.2. A megszakításfeldolgozás hardver része ..... 23
2.3. A megszakításfeldolgozás szoftver része ..... 25
2.4. A megszakításrendszerek típusai ..... 27
2.4.1. Az egyszintű megszakításrendszer ..... 27
2.4.2. A kétszintű megszakításrendszer ..... 28
2.4.3. A többszintű megszakításrendszer ..... 30

**3. Az operációs rendszer elindítása és leállítása**
3.1. IPL az IBM 370, 390 gépcsaládokon ..... 33
3.2. Bootolás az IBM PC-n ..... 35
3.3. Az operációs rendszerek leállítása és újraindítása ..... 36

**4. A processzorgazdálkodás módszerei**
4.1. A prioritásos módszer ..... 39
4.2. Az időszeleteléses módszer ..... 40
4.3. Kombinált módszerek ..... 40

**5. A memóriagazdálkodás módszerei**
5.1. A memória fix felosztása ..... 43
5.2. A memória dinamikus felhasználása ..... 44
5.3. A virtuális memória ..... 45
5.4. A VIO, avagy a memória-diszk ..... 49
5.5. A pufferezés ..... 50

**6. A periféria-gazdálkodás**
6.1. A párbeszédes és a kötegelt feldolgozás ....55
6.2. A spool-rendszerek ....56
6.3. Fájlrendszerek ....58

**7. Többprocesszoros rendszerek**
7.1. Egy memória - több processzor ....67
7.2. Több memória - több processzor ....68
7.3. Többgépes rendszerek ....69
7.4. Számítóháló (Rács, GRID) ....71

**8. Párhuzamosság, szinkronizálás**
8.1. Programvezérelt megoldások ....77
8.2. Automatizált megoldások ....78
8.3. Kliens-szerver megoldások ....80

**9. Holtpontmentes vezérlés**
9.1. A holtpont ....83
9.2. A holtpont kialakulásának felismerése ....85
9.3. A holtponti helyzet megoldása ....86
9.4. A holtponti helyzet elkerülése ....87
9.5. Erőforrások feltételes megszerzése ....90

**10. Kommunikáció**
10.1. Az operátori parancsok ....95
10.2. Parancsnyelvek ....95
10.3. A felhasználó kommunikációja ....96
10.4. A rendszerprogramozó parancsai ....96
10.5. A kommunikáció kiszolgálása ....96
10.6. A kommunikáció stílusa ....97

**11. Alkalmazások készítésének eszközei**
11.1. Fordító programok ....99
11.1.1. ASSEMBLERek ....100
11.1.2. Magasszintű programozási nyelvek ....100
11.2. A szerkesztés ....100
11.3. Alkalmazásgenerátor ....102
11.4. Programgenerátorok ....103
11.5. Az interpreterek ....103
11.6. A JAVA módszere ....104
11.7. Segédprogramok ....104

**12. Operációs rendszereket kiszolgáló operációs rendszerek**
12.1. A virtuális gép ....107
12.2. Hálózati operációs rendszerek ....110

**13. Az operációs rendszer installálása, hangolása**
13.1. Amikor nincs operációs rendszer, a stand-alone programok .....113

**14. Virtuális megoldások**
14.1. Virtualitások.....115
14.2. Felhő .....118

**Zárszó** .....121
**Tárgymutató** .....123
**Irodalomjegyzék** .....127

# *ELŐSZÓ*

Ezen jegyzet a Dunaújvárosi Egyetemen tartott előadásaim alapján készült.

A jegyzet az általános célú számítógépek operációs rendszerei* szokásos működési elveivel és az alkalmazott főbb módszerek tárgyalásával foglalkozik.

A jegyzetben feldolgozott témák tárgyalásakor a számítógépekkel kapcsolatban bizonyos előismeretek meglétét feltételezzük, azokra építünk. Ezek az ismeretek például a számítógépek architektúrája kurzusokon megszerezhető ismeretek.

Ezúton is köszönetet mondok **Kógelmann Gábor**nak és **Molnár László**nak a kézirat gondos átnézésében nyújtott segítségéért, hasznos kiegészítéseikért, tanácsaikért.

Dunaújváros, 2016. március

*Buza Antal*

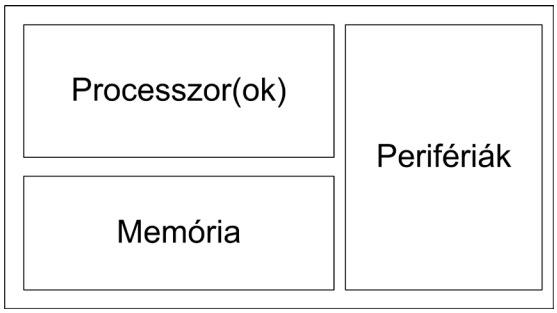
* Az „operációs rendszer”-t gyakran egybeírva, „operációsrendszer”-ként használják. E jegyzetben az MTA SzTAKI szótárában található írásmódnak megfelelően két szóba írjuk.

# *1. BEVEZETÉS, AZ OPERÁCIÓS RENDSZEREK KIALAKULÁSA*

## 1.1. Az egyszerű számítógép-modell

A mindennapi gyakorlatban analóg, digitális és hibrid számítógépeket használnak. Az analóg és a hibrid számítógépek rendszerint valamely célfeladatra tervezett, egyedi jellegű számítógépek. Ezeknek nemcsak szerkezetét, hanem működését is előre pontosan megtervezik, s az alig változtatható. Az általános célú, digitális számítógépeket előre pontosan nem-meghatározott felhasználási területeken, igen változatos feladatok megoldására építik. Ahhoz, hogy az előre pontosan nem ismert feladatok (akár párhuzamos) megoldására is alkalmasak legyenek, vált szükségessé az ezen számítógépek munkáját irányító, felügyelő, megszervező programrendszerek, az operációs rendszerek kidolgozása. Így tehát a továbbiakban csak általános célú, digitális számítógépekkel foglalkozunk.

A számítógépeket sokféleképpen lehet meghatározni, jelen meggondolásaink szempontjából az *1.1. ábrán* látható – igen egyszerűsített – modellt fogjuk használni.




*1.1. ábra. Az egyszerű számítógép-modell.*

Ez sok tekintetben egyszerűsített, elnagyolt modell. A felhasználók rendszerint egy számítógéprendszerrel dolgoznak, melyben sok és sokféle memória, processzor és periféria működik. Ezt kívánja például az IBM azzal is érzékeltetni, hogy nagy számítógépeit „Processor Complex”-nek nevezi.

A számítógépekben gyakran alkalmazott módszer az is, hogy bizonyos funkciókat (rendes körülmények között a felhasználó által) nem módosítható, (többnyire a hardver gyártója által) előre elkészített programmal valósítanak meg. Ezek a programok tipikusan valamely hardvereszköz közvetlen működtetésében vesznek részt, készítésük, módosításuk sokkal inkább hardvermérnöki, mint programozói feladat. Az ilyen programokat általában ROM-ban (Read Only Memory, azaz csak olvasható memória) helyezik el. Minthogy ezen programokat (rendes körülmények között) nem módosíthatjuk, így a számítógép adottságának kell tekintenünk, és ezért az operációs rendszerek tanulmányozása szempontjából a hardverhez soroljuk őket. Ez megfelel annak a szóhasználatnak is, amikor gépi utasításokról beszélünk. A mi szempontunkból „gépi utasítások” azok, amelyekből a futtatható (lefordított, megszerkesztett) programok állnak.

A valóságban a gépi utasítás végrehajtása sok esetben egy (vagy több) mikroprogram végrehajtását jelenti, ezt azonban programozóként nem érzékeljük.

A mai számítógépek az úgynevezett **„tárolt programozás”** elvén működnek. Ez azt jelenti, hogy a feladat elvégzéséhez szükséges következő utasítást nem egy-egy utasítás befejezése után adjuk meg, mint ahogy azt például az egyszerűbb kalkulátoroknál tesszük, hanem előre megadjuk a teljes utasítássorozatot, azaz a programot. Ezzel azt érjük el, hogy a feladat elvégzéséhez (tehát az összes éppen szükséges utasítás végrehajtásához) szükséges idő csak a számítógépünk alkatrészei sebességén múljon, és nem kell a gépünknek várakoznia a soron következő utasításra, azaz a gépi sebességhez képest rendkívül lassú emberre. Az előre megadott utasítássorozatot valahol olyan állapotban kell tárolni, amely a számítógépünk számára automatikusan olvasható. Lényegében ezért jelennek meg a memóriák.

Bár természetesen a kalkulátorok is rendelkeznek valamilyen tárolóképességgel, hiszen az egyes műveletek operandusai értékét, és a kiszámított eredményt legalábbis ideiglenesen tárolják, az ilyen tároló „rekeszeket” általában a processzor részeként valósítják meg és a szóhasználatban nem is memóriának, hanem regisztereknek nevezik őket.

A **memória:** valamilyen fizikai hatásra (mágnesezettségi állapot, töltöttségi állapot, feszültségszint, fény, stb.) épülő tárolókészülék. Mivel nagy biztonsággal, viszonylag egyszerűen, gyorsan, két állapot különböztethető meg (van fényjelenség – nincs fényjelenség, a mérhető feszültség egy határértéknél kisebb-nagyobb, a mágnesezettségi állapot ilyen-vagy olyan áramot indukál, stb.) így általában kétállapotú rendszereket használunk adattárolásra. Ezek a digitális tárolók. A két állapot jelölésére a 0-át és az 1-et használjuk. A tárolt adat tehát 0-ákból és 1-esekből álló sorozattal írható le. Egy ilyen számsorozatot bináris számnak tekinthetünk. Ebben az értelemben beszélünk arról, hogy a memóriában az adatokat (és a programokat, de ezek is adatok, csak a jelentéstartalmukat tekintve utasításokként értelmezzük őket) bináris formában tároljuk.

A tárolt programozás elvének megfelelően a futtatandó programot tehát a memóriában kell elhelyezni. Ahhoz, hogy a programot futtatni tudjuk, azaz utasításait a program által meghatározott sorrendben a processzor „automatikusan” végrehajthassa, szükséges olyan mechanizmus kialakítása, mely segítségével mindig megállapítható a következőként végrehajtandó utasítás (kódjának) memóriabeli helye. Ezt címszámláló regiszterrel oldják meg. Az, hogy melyik a következő végrehajtandó utasítás, függhet az utolsóként végrehajtott utasítás végrehajtásának eredményétől és eredményességétől. (Megeshet például, hogy az utasítást nem lehetett végrehajtani, például ha az utolsóként végrehajtani szándékozott utasítás osztás volt, és osztójának aktuális értéke nulla volt.) Szükséges tehát, hogy az utoljára végrehajtott (illetve megkísérelt) utasítás végrehajtásának eredményéről információt tároljunk. Ezt jelzőbitek beállításával oldják meg. A tárolt programozás megoldásához tehát az eredetileg egyszerű számítógép-modellt néhány műszaki megoldással ki kell egészíteni.

A tárolt programozás és a soros utasítás-végrehajtás elvét a magyar származású **Neumann János (1903–1957)** által vezetett kutatócsoport dolgozta ki, ezért sokszor **„Neumann-elv”**-ként is hivatkoznak a tárolt programozás ötletére. Ennek megfelelően az ilyen számítógépeket **Neumann-elvű számítógép**eknek is nevezzük. Hosszabb ideje folynak kutatások és kísérletek más, ettől eltérő működési módú számítógépek megalkotására is, meglehet, hogy majd megjelennek új elven működő számítógépek is. A ma használatban levő általános célú számítógépek lényegében mind Neumann-elvű számítógépek. Ezért a továbbiakban csak a Neumann-elvű számítógépek operációs rendszereivel foglalkozunk.

### 1.2. Kihasználatlanságok

Vannak olyan számítógépek, melyeket egy (vagy néhány) előre pontosan ismert feladat megoldására építenek. (Ezeket például az automatákban, folyamatszabályozásban használják.) Mi a továbbiakban nem ezen „célgépekkel", hanem az univerzális számítógépekkel foglalkozunk. Ahogy ez a megnevezésből is következik ezen számítógépeknek változatos feladatok megoldására kell alkalmasnak lenniük. A számítógép használata programok végrehajtását, „futtatását" jelenti. A tárolt programozási elven működő számítógépen a program végrehajtásakor a programnak (de legalábbis egy részének) a memóriában kell lennie. Különböző feladatokat megoldó programok mérete is természetesen különböző. Így nyilvánvalóan nem lehet pontosan megfelelő méretű memóriával építeni a számítógépet. Azaz, az egyetlen programmal működtetett univerzális célú számítógép memóriájában (egészen ritka kivételtől eltekintve) mindig van „éppen felesleges", szabad hely. Másképpen fogalmazva: **egyetlen program futtatása esetén a memória szinte soha nincs teljesen kihasználva.**

Vizsgáljuk meg a processzor kihasználtságát is. A számítógépen futó program a memóriában helyezkedik el, utasításait pedig a processzor hajtja végre. Az esetek nagyon nagy részében a program működéséhez adatok szükségesek, másrészt az eredményeket sem elég csak a memóriában létrehozni. Az eredményeket vagy további számítógépi felhasználás céljából valamilyen adattárolón kell elhelyezni, vagy emberi (esetleg más) felhasználás céljának megfelelő alakra kell hozni. (Papírra kinyomtatni, képernyőn megjeleníteni, hangot generálni, szelepet elzárni stb.) A számítógépipar fejlődésével a processzorok utasítás-végrehajtási sebessége rendkívüli módon megnőtt. A perifériák (nagy részének) sebessége is nőtt, de messze alatta marad a processzor sebességnek. Ez oda vezet, hogy amikor a programban (például) egy input/output (adatbeviteli – adatkiviteli) művelethez érkezünk, akkor a processzor ezen műveletet elindítja, és (mint rendesen) a következő utasítás végrehajtásával mindaddig várakozik amíg az elindított művelet befejeződik. Mivel a perifériák lassúak (a processzorhoz képest) így a processzor idejének nagyon nagy részét tétlenül, várakozással tölti. Különösen így van ez a párbeszédes programok esetében. Amikor a párbeszédes program feltesz egy kérdést, ettől kezdve a program mindaddig várakozik, amíg a választ meg nem kapja. Ha azonnal válaszol is a felhasználó, az is 1–2 másodpercig tart, azaz a processzor ez idő alatt több millió (10–100 millió) műveletet is elvégezhetett volna. A helyzet csak romlik, ha nem válaszolunk azonnal, hanem valaminek utána kell nézni, valamit valakitől meg kell kérdezni, vagy bármely más okból percek is eltelnek. Általában azt tapasztaljuk tehát, hogy **egyetlen programmal működtetve a számítógépet, annak processzora általában alig kihasznált.**

Fentieket összefoglalva arra a megállapításra jutunk, hogy egy számítógépet egyetlen programmal működtetve két „legfőbb" részének, a processzornak és a memóriának a kihasználtsága alacsony, illetve igen alacsony. Másrészt igen tipikus (több felhasználót kiszolgáló számítógépeknél szinte mindig, de a személyi számítógépek esetében is sokszor így van) az a helyzet, hogy egy adott időben nem csak egyetlen feladat megoldására szeretnénk a számítógépet használni. Ha a gépünk egyszerre csak egyetlen program futtatására alkalmas, akkor nyilvánvaló, hogy csak az egyik program befejeződése után indítható el a következő program. Ez különösen a sokfelhasználós rendszerek esetében nem felelne meg a felhasználók igényeinek, hiszen egyetlen felhasználót kivéve mindenkit várakozásra kényszerítene.

### 1.3. Az ötlet

Számos más meggondolás mellett a most tárgyalt fő okok vezettek a következő ötlethez:
– Minthogy az éppen futó program mellett a memóriában általában úgyis marad szabad hely, töltsünk be oda egy másik végrehajtandó programot,
– Minthogy a processzor a program futtatása közben sokszor és (a sebességéhez képest) sokat várakozik, foglalkozzon e várakozási időkben a másik betöltött program utasításai végrehajtásával.
– Természetesen ez nemcsak egy második programra vonatkozik, hanem harmadik, negyedik, stb. programok is „bekészíthetők" a memóriába.

Ezen ötlettel a felmerült gondok mindegyike (legalább részben) megoldódik. A memória kihasználtsága nyilvánvalóan javul, csak akkora kihasználatlan hely marad, amekkora a még elindítani kívánt programok egyike számára sem elegendő. A processzor kihasználtsága is jelentősen nő, hiszen amikor az éppen végrehajtott program várakozni kényszerül, akkor egy másik program utasításaival foglalkozhat, s ha itt is várakozásra kényszerül (és az első program is még mindig várakozó állapotban van) akkor egy harmadik programmal dolgozhat, és így tovább. Az ötlet tehát igen jó(nak látszik). A most ismertetett ötletet nevezik a **multiprogramozottság elv**ének.

### 1.4. Megvalósítható-e?

Az egyszerre egyetlen program működtetésére tervezett számítógép azonban ilyen működésre alkalmatlan. A programok futásának eredménye nyilván nem függhet attól, hogy milyen más programokkal együtt került éppen végrehajtásra. Másképpen fogalmazva gondoskodni kell arról, hogy a programok semmiképpen se zavarják, befolyásolják egymás működését. A programok egyszerre használják a számítógép memóriáját, váltakozva a processzort és a perifériákat.

Tekintsük át a multiprogramozott működés okozta főbb problémákat:
– Amíg a számítógépen egyetlen program működik, addig az (helyesebben a program írója) a teljes memóriával szabadon gazdálkodik. Oda helyez el utasítást, vagy adatot, ahova csak kedve tartja. Ha a multiprogramozott ötletet akarjuk megvalósítani, akkor ez a helyzet alapvetően megváltozik. Ha a programozók továbbra is a teljes memóriát használhatnák, az egymás adatainak és/vagy utasításainak a felülírását okozhatná, ami megengedhetetlen, mert így a jól megírt programok működése teljesen esetlegessé, bizonytalanná válna. Nyilvánvalóan gondoskodni kell tehát arról, hogy minden működő program rendelkezzen olyan védett memóriaterülettel, amelyhez a többi futó programok nem férhetnek hozzá. E feladatot röviden **memóriavédelem** néven emlegetik.
– A multiprogramozás ötletei közé tartozik az, hogy az első program által nem használt memóriaterületen helyezzük el a második futtatni kívánt programot. Hol van ez a bizonyos szabad hely a memóriában? Ha az első program kis memóriaterületet igényel, akkor már a memória elejéhez közel kezdődik a szabad hely, ha az első program nagy memóriaterületet igényel, akkor csak később kezdődik az a szabad hely, ahová a második program kerülhet. Miért probléma ez? Azért mert az eredeti tárolt programozású számítógép gépi utasításai olyanok voltak, hogy minden (gépi) utasítás tartalmazta az utasítás kódját (azt, hogy mit is kell csinálni) és az utasításban érintett adatok memóriabeli címét. Ez az egy programmal működő számítógépre alkalmas megoldás volt. Ha azonban a második program (és bármelyik program lehet éppen második is) olyan memóriaterületet használhat, melynek helyét a program írásakor előre nem ismerjük, akkor olyan módszerre van szükség, amely módszer lehetővé teszi, hogy a programot tetszőleges memóriaterületre is elhelyezhessük, s az ekkor is helyesen találja meg az utasításokban érintett adatokat,

amelyek pontos helye szintén függ attól, hogy a program melyik memóriaterületre kerül. Meg kell tehát oldani azt, hogy **a programok, adatok más-más memóriacímtől történő elhelyezése lehetséges legyen** és ez természetesen ne befolyásolja a jól megírt program helyes működését.
– A multiprogramozás elgondolásának része volt a következő is: „minthogy a processzor a program futtatása közben sokszor és (a sebességéhez képest) sokat várakozik, foglalkozzon e várakozási időkben a másik betöltött program utasításai végrehajtásával". De mi történjen akkor, ha az az esemény, amelyre valamelyik program várakozott, bekövetkezett. Ekkor a processzor valamelyik program utasításait hajtja végre, s közben folytathatóvá vált egy másik program is. Valahogy el kell tehát dönteni, hogy a futtatható állapotú programok közül a processzor melyik program utasításainak végrehajtásával foglalkozzon. Ezt a feladatot röviden a **processzoridő-kiosztás**ának nevezzük.
– Érzékelhető, hogy nyilván kell tartani azt, hogy melyik program futtatható éppen, és mely programok mire várakoznak. Amikor pedig egy várt esemény bekövetkezik, akkor ezt a tényt a nyilvántartásban követni kell, például azzal, hogy az ezen eseményre korábban várakozó program már folytatható állapotba került. Az esemény bekövetkezésének időszakában azonban a processzor valamely más program futtatásával foglalkozott. Meg kell tehát oldani azt, hogy események bekövetkeztekor az éppen futó program futtatása ideiglenesen álljon meg, azért, hogy például a nyilvántartást módosíthassuk, de a megállított program később folytatható legyen méghozzá úgy, hogy a program működésében semmilyen változás – emiatt az ideiglenes megállítás, majd folytatás miatt – ne történjen, a megállás „észrevétlen maradjon". Ezt a feladatot összefoglalóan a **futó program bármikori megállításának és továbbindításának megoldása** néven nevezzük.
– Ha multiprogramozottan két program fut, és mindkettő például nyomtatni kíván teljesen használhatatlan lenne az olyan lista, melyben néhány sort az egyik, aztán – megjósolhatatlan számú – néhány sort a másik program ír és így tovább. Ez a példaként felhozott probléma persze áthidalható lenne több nyomtató működtetésével, de egyrészt ez korlátozná a multiprogramozottság szintjét, másrészt rendkívül gazdaságtalan lenne. A nyomtatók számát, teljesítményét a nyomtatási igényekhez és nem az együttfutó programok számához kell illeszteni. Minden egyéb perifériára vonatkozóan hasonló helyzetek fordulhatnak elő. Ezt a problémakört összefoglalóan perifériakezelésként tárgyaljuk.

Fentieket összefoglalva tehát a multiprogramozott működés megvalósításához szükséges feltételek egy része:
– a memóriavédelem,
– a programok, adatok más-más memóriacímtől történő elhelyezésének lehetősége,
– a processzoridő kiosztása,
– a perifériakezelés,
– a futó program bármikori megállításának és továbbindításának megoldása.

A program működéséhez szükséges, a program által használható eszközöket **erőforrás**oknak is szokás nevezni. (Erőforrás a processzor, illetve vele kapcsolatban, bizonyos értelemben az idő is, a memória és a perifériák, erőforrásnak tekinthetők olyan, szolgáltatásokat nyújtó programok is, amelyek szolgáltatásait egy vizsgált program igénybe veszi.) E szóhasználattal élve a multiprogramozott működés megvalósításához szükséges az erőforrásokkal való gazdálkodás megoldása.

### 1.5. A megoldás

A megoldandó feladatok egy részére hardver-megoldás a célszerű. Például a programok másmás memóriacímre való betölthetőségének egyik megoldásához a gépi utasításokban olyan címzési módszert (bázisregiszteres címzés) választottak, mely megfelelő értelmezéséhez a hardvert is (esetleg „csak" a mikroprogramot) alkalmassá kell tenni. A memóriavédelem megoldására is hardver-módszereket alkalmaznak, ez ugyan (első ránézésre talán) szoftverrel is megoldható lenne, de mégis a hardvermegoldás a célszerűbb.

Ennek két fő oka az, hogy a memóriavédelem nem igényel bonyolult döntéseket, algoritmusokat, hiszen csak azt kell eldönteni, hogy adott program az adott memóriacímhez hozzáférhet-e vagy sem, másrészt minden egyes gépi utasítás végrehajtásakor (utasítások nagy többségében többször is) figyelni kell a memóriavédelemre. Ha minduntalan egy programot kellene emiatt működtetni, az rendkívül terhelné a processzort, nagyon rontva ezzel a felhasználó által érzékelt teljesítményét, nem is beszélve arról, hogy ezen ellenőrző program utasításait is ellenőrizni kell ugyanezen programmal, ami nyilvánvalóan végtelen ellenőrzési mélységhez vezetne – ez nem volna szerencsés megoldás.

A feladatok más részének megoldására, ahol döntéseket kell hozni, azaz algoritmusokat kell alkalmazni, megfelelő döntéshozó-vezérlő programok, vagyis szoftver az alkalmas.

### 1.6. A megoldás szülte újabb feladatok és megoldásaik

Abban az esetben, ha a számítógép működését programokkal irányítjuk-szervezzük, akkor egyrészt

– meg kell oldani azt, hogy ez a program(rendszer) minden eseményről feltétlenül értesüljön.

Erre, mivel (az egyszerű modellünkben) az egyetlen processzorunk csak egyetlen program egyetlen utasításával foglalkozik megint csak hardvermegoldás lehet alkalmas.

Így alakult ki a **megszakításrendszer**, vele bővebben a *2. fejezet*ben foglalkozunk.

Másrészt

– meg kell oldani, hogy a vezérlő-irányító programot a felhasználói programok (és általában semmilyen program) „ne kerülhesse meg", azaz bizonyos feladatokat csak a vezérlőprogram tudjon elvégezni, tehát bizonyos utasításokat csak a vezérlőprogram használhasson.

Így alakultak ki a **privilegizált utasítások**. A gépi utasításokat két csoportba sorolják. Az egyik csoport a normál utasítások csoportja, a másik csoport a privilegizált utasítások csoportja. A processzoron megjelenik egy jelzőbit, amellyel a processzor két állapotát különböztetik meg. A processzor normál állapotában a normál utasításokat végrehajtja, a privilegizált utasításokat nem. A processzor privilegizált állapotában mind a normál, mind a privilegizált utasításokat végrehajtja.

Az operációs rendszer privilegizált állapotban működik, a többi programok végrehajtásakor a processzor normál állapotban van. Így elérhető, hogy ha a programozó ismeri is a privilegizált utasításokat, és azokat használja is a programjaiban, azok ekkor sem fognak végrehajtódni, tehát a speciális funkciókat csak az operációs rendszer programjai tudják használni. A processzor állapotváltása (normálról privilegizáltra és vissza) szorosan összefügg a *2. fejezet*ben tárgyalt megszakításrendszerrel.

Továbbá, ha a gép működését (ami programok futtatását jelenti) egy programrendszer irányítja, azaz minden program ennek irányításával töltődik a memóriába, fut és fejeződik be, akkor

– meg kell oldani a vezérlő-irányító programrendszer **betöltését és elindítását**.

Minthogy ezt – legalábbis legelső fázisait – még nem felügyelheti program, így hardvermegoldást lehet csak alkalmazni. Ez azonban nem új feladat. Ugyanezt az egyetlen program futtatására alkalmas számítógépen is meg kellett oldani, hiszen a futtatandó programot ott is be kellett tölteni a memóriába és meg kellett oldani elindítását is. Az e feladatra korábban kialakított módszer most is alkalmazható.

*Összefoglalva:* az egyszerre egyetlen program kiszolgálására alkalmas egyszerű számítógép nagyon rosszul kihasznált. A kihasználtságot jelentősen javítja a multiprogramozott működtetés. Ennek megvalósításához részben új hardverfunkciókkal kell bővíteni az egyszerű számítógépmodellt, részben a számítógép munkáját irányító programrendszert kell készíteni.

*A számítógép multiprogramozott felhasználásával kapcsolatban vannak szoftverrel megoldandó feladatok is. Azt a programegyüttest, amely a multiprogramozott számítógéphasználatot lehetővé teszi, a számítógép ilyen működését felügyeli, irányítja, koordinálja,* ***operációs rendszer****nek, vagy* ***műveletvezérlő rendszer****nek nevezzük.*

A számítógép használatával a cél a felhasználói igény kielégítése, így a számítógépen általában valamely felhasználói program fut. Az egyszerű modell egyetlen processzora egyetlen program egyetlen utasításával foglalkozik egy adott pillanatban. Ha ez felhasználói program, akkor ezalatt az operációs rendszer nincs működésben, legfeljebb egyes részei a memóriában vannak. Amikor egy program utasításai fennakadás, és az operációs rendszer szolgáltatásai nélkül végrehajthatóak, nem is kell az operációs rendszernek működnie. Abban az esetben kell az operációs rendszernek működnie, ha a program futása valamilyen okból megakad (például az I/O igények kielégítésére vár) vagy ha az általa irányított számítógéprendszerben valamilyen rendszerkiszolgálást igénylő helyzet áll elő. Az éppen futó program megállítására és az operációs rendszer (szükséges moduljai) elindítására a **megszakítás rendszer**nek nevezett hardver–szoftver-mechanizmust alkalmazzák. Ehhez célszerű, hogy bizonyos memóriaterületen az operációs rendszer rutinjai (programjai) elindítható-, állapotleíró táblázatai pedig használható állapotban jelen legyenek.

Ha a gép működése közben a memóriában az operációs rendszer bizonyos moduljai állandóan (rezidensen) jelen vannak, és „bármikor" elindíthatóak, elég kézenfekvő az az ötlet is, hogy az operációs rendszer az eddig tárgyalt – a felhasználót csak közvetetten szolgáló – feladatokon kívül a számítógép kezelőjének, a programozójának és a felhasználójának is közvetlen szolgáltatásokat nyújtson. Ezek a munkát, az eligazodást segítő vagy egyéb „kényelmi" szolgáltatások. Legtágabb értelemben az operációs rendszerekhez szokták sorolni a felhasználói igényt közvetlenül kielégítő programokon kívül az összes egyéb programokat (fordítók, interpreterek, szerkesztők, segédprogramok, stb.). A határ a felhasználói és rendszerprogramok között nem egyértelműen adható meg. A jegyzet első tíz fejezetében a szűkebb értelemben vett operációs rendszerekkel foglalkozunk, nem számítjuk ide a számítógéprendszer működtetéséhez nem feltétlenül szükséges külön, önálló programtermékként beszerezhető programokat, programcsomagokat.

Az operációs rendszer tehát a számítógép munkáját irányítja, algoritmusaival döntéseket hoz. Minthogy az univerzális számítógép felhasználási területei nagyon különbözőek, így nem várható el egy előre rögzített algoritmussal működő operációs rendszertől, hogy az minden felhasználási környezetben a legalkalmasabb gépkihasználást biztosítsa. Ezért az operációs rendszerek sok működési paramétere az üzembehelyezésekor, nagy részük a rendszer elindításakor is, néhány pedig működés közben is megváltoztatható. Ahhoz, hogy működése közben a rendszer paramétereit megváltoztathassuk, a rendszerrel kommunikálnunk kell. Ezért az operációs rendszerek

egyik természetesnek tekintett funkciója a kezelővel való kommunikáció is. Az operációs rendszert (és általa a számítógéprendszert) a számítógépkezelő párbeszédes módon irányítja. Természetesen nemcsak paramétereket állíthatunk be gépkezelői parancsokkal, hanem információs parancsokkal a rendszer állapotáról, a futó, a várakozó programokról, a memória és a perifériák állapotáról, esetleg a hálózat állapotáról, általában minden fontos jellemzőjéről tájékozódhatunk. A rendszer egy sor kényelmi szolgáltatást is biztosít(hat).

Külön szólni kell a *programok és adatok védelméről*. A felhasználó számára az adat igen nagy értéket jelenthet. Meglehet, hogy több év, sok munka van az adatok létrehozásában, meghibásodásuk a felhasználónak nagy gondot és kárt okozhat. Ezért a felhasználónak nyilvánvaló igénye, hogy a programjaival együttfutó más programok ne tehessék tönkre az ő adatait, de illetéktelenül még csak hozzájuk se férhessenek. Sok esetben magát a géphez való hozzáférést is jogosultsághoz kötik. Ezen védelmi feladatokra is alkalmasnak kell lennie az operációs rendszernek.
Nemcsak a véletlen vagy szándékos károkozásra kell számítani, hanem az alkatrészek meghibásodására is. Ha alkatrész (hardver) hibásodik meg, az bár kiesést és költséget jelent, de pótolható, megvásárolható. Ha szoftver hibásodik meg (előfordulhat, például vírusok tönkretehetnek korábban jó programokat is) az is kiesést jelent, de pótolható, mert a gondos rendszergazdától elvárható, hogy az üzemeltetett szoftverek installáló anyaga rendelkezésére álljon. Ha azonban adat megy tönkre, az egyszerűen nem pótolható. Nem tudunk elmenni a boltba és vásárolni (ahogy a hardver és végül a szoftver esetében ez lehetséges). Az adat tönkremenetele problémáját rendszeres mentéssel sem tudjuk kivédeni, legalábbis olyan rendszerek esetében, amelyek adattartalma gyakran változik (például a Neptun, vagy a banki, raktári, kórházi, stb. folyamatos működésű rendszerek, melyek adatai lényegében állandóan változnak). Így a helyreállítás lehetősége a mentés és naplózás közös, összehangolt használatát igényli, komplex és fontos feladat. Elvárjuk, az operációs rendszerektől, hogy *mentési-visszaállítási* szolgáltatásuk is legyen.

A mentési-helyreállítási feladatot az operációs rendszer önmagában nem képes teljes biztonsággal működtetni, ahhoz a rendszergazda előrelátó és aktív közreműködése is szükséges. Egy, a számítógép közelében tartott adatmentés elegendő lehet például a mágneslemez tönkremenetele után a működőképesség újbóli eléréséhez. De a számítógép közelében tartott adatmentés semmire nem jó, ha elemi csapás (tűz, árvíz, földrengés, és hasonlók) fordulnak elő, hiszen ekkor várhatóan a mentés is megsemmisül. Az elvárható gondosság fogalomkörébe tartozik, hogy ilyen esetekre felkészülve a mentést (vagy annak másolatát) nagy földrajzi távolságokban tartsuk. Annak valószínűsége, hogy a több helyen, nagy távolságban tartott mentésekkel is ugyanakkor történjen valami katasztrófa, remélhetőleg kicsi. A nagy földrajzi távolság technikailag már nem probléma, a hálózatok gyors és olcsó működésűek. Természetesen a hozzáférés-biztonsági kérdések ilyenkor fokozottan jelentkeznek. Ennek részletes tárgyalása túlmutat e jegyzet keretein.

A számítógépekkel (többnyire a közepes és nagy számítógépeken) sok egymástól független felhasználó dolgozik. A gép költségeit a felhasználás arányához igazodóan szét kell tudni osztani. Különösen fontos ez, ha a számítógépet szolgáltatásjelleggel működtetik. Minthogy minden program az operációs rendszer felügyelete alatt működik, így az operációs rendszer mérni és gyűjteni tudja az erőforrások felhasználási adatait (processzorhasználati idő, memóriahasználat jellemzői, periférián használt hely, nyomtatott sorok-lapok száma, stb.). Az operációs rendszer ilyen szolgáltatását *elszámolási* (account) rendszernek nevezik.

Sokszor felvetődik az a kérdés, hogy vajon mikor mi történt a rendszerben? Ki, mikor, hol (honnan), mit tett, vagy kísérelt meg tenni? Ezen kérdések összefüggenek a biztonsági, az el-

számolási, hibakeresési, helyreállítási feladatok megoldásával. E célból az operációs rendszerek a történéseket egy (több) naplófájlba rögzíthetik, ezt a tevékenységet *naplózás*nak nevezzük.

## 1.7. Az operációs rendszerek feladatai

Az eddigieket összefoglalva az operációs rendszerek feladata tehát:
- a felhasználói programok **működési körülmény**einek biztosítása, igényeik kiszolgálása,
- **gazdálkodás** a számítógéprendszer erőforrásaival, azaz a processzorokkal, a memóriákkal, a perifériákkal,
- **kommunikáció** a gépkezelővel*, a programozóval, esetleg a felhasználóval, valamint felhasználó-program, program-program, operációs rendszerfelhasználói program, operációs rendszeroperációs-rendszer kommunikációk lehetővé tétele,
- az **adatvédelem** és a **programvédelem** megoldása,
- a **naplózási tevékenységek** elvégzése,
- **mentési-helyreállítási** tevékenység lehetősége,
- **elszámolási** rendszer működtetése,
- egyéb szolgáltatások (**segédprogramok**) biztosítása a gépkezelő, a programozó és a felhasználó számára.

* Gépkezelő, vagy operátor az, aki a számítóközpontokban a számítógépek üzemeltetésével, kiszolgálásával foglalkozik. Ilyen munkakört inkább csak nagyrendszerek üzemeltetésekor szoktak létesíteni. A személyi számítógépeknél a felhasználó és az üzemeltető sok esetben ugyanaz az egyetlen személy.

## 1.8. A számítógép működése operációs rendszerrel

Az operációs rendszer felügyelete alatt működő számítógépen rendesen általában felhasználói program fut (hiszen ezért használják a számítógépet). Ha döntési feladat, vagy egyéb, a rendszer által kiszolgálandó igény áll elő, akkor működik az operációs rendszer. Az igény mindig megszakítás formájában jelentkezik. Ekkor a hardver megszakítja az éppen futó felhasználói program végrehajtását, és elindul az operációs rendszer megfelelő rutinja. Az igény kiszolgálása után valamely egyébként futtatható állapotban levő program utasításaival foglalkozik ismét a processzor, egészen a legközelebbi megszakításig. Ez ugyan a felhasználó által nem közvetlenül megkívánt programok (az operációs rendszer moduljai) működését is eredményezi, a gépet (legfőképp a processzort) terheli, de az operációs rendszer működése egyrészt elengedhetetlenül szükséges, másrészt a processzor szabadidejébe többnyire „bőven" belefér, jól kiegyensúlyozott esetben nem hátráltatja lényegesen a felhasználói programok futását. Ha kis teljesítményű gépen bonyolult operációs rendszert, vagy egyszerre nagyon sok programot akarunk futtatni, akkor előállhat olyan helyzet, hogy túlnyomórészt az operációs rendszer működik és a felhasználói programok nem, vagy alig haladnak. Ekkor vagy egyszerűbb (ezzel persze kevesebbet tudó) operációs rendszert kell használni, vagy csökkenteni kell a rendszer terhelését, vagy növelni kell a hardver teljesítményét. A jobb operációs rendszerekben lehetőségünk van mérni a számítógéprendszer egyes részeinek programok általi kihasználtságát, ezzel (is) segítséget adva a szűk keresztmetszetek felderítésére.

## 1.9. Az operációs rendszer felépítése, komponensei

Az operációs rendszer főbb komponensei:
– **programok:** megszakítás-feldolgozó programok (az igénykiszolgáló, eseményfeldolgozó és a döntéshozó algoritmusokat megvalósító programok), valamint további a kezelést, használatot segítő segédprogramok,
– memóriában tárolt **táblázatok** és esetleg más célszerű adatszerkezetek: (melyekben a pillanatnyi állapotokat, és az algoritmusai által igényelt paramétereket tartja nyilván),
– **fájlok**, a rendszer elindításához és működéséhez szükséges program-, paraméter- és adatfájlok.

A háttértárolókon a programokat és adatokat is fájlokban tároljuk. A fenti komponens-összefoglalóban logikailag, funkciójuk szerint, különböztetjük meg őket. Fájl alatt a fenti harmadik pontban nem a programokat, hanem az operációs rendszer működéséhez szükséges adatokat tartalmazó fájlokat értjük.

Az operációs rendszerek felépítésük szerint is több nagy csoportba sorolhatók. Az egyik csoport a **monolitikus** rendszerek csoportja. Ekkor az operációs rendszert szinte egyetlen programként valósítják meg (mely természetesen azért sok részből állhat). Ezek erősen integrált rendszerek. Hibajavításuk és fejlesztésük nehézkes, az egész rendszer működésének áttekintését igényli, nagyobb rendszereknél alig megoldható feladat. Igen nehezen követhető, hogy a rendszerben valahol történő változtatásnak másutt milyen következményei lesznek. Sokak véleménye szerint nagy és bonyolult rendszereket nem helyes monolitikusra tervezni-megvalósítani. Monolitikus jellegű rendszerek a DOS, Win95–98-ME.

**Moduláris** szerkezetű rendszerek azok, melyek egymástól jól elkülöníthető, nagyon pontosan definiált kapcsolatokkal és szolgáltatásokkal működő modulokból épülnek fel. Minden modul jól definiált paraméterezéssel használható, az előre megadott szolgáltatásokat biztosítja. Szokás ezt a felépítési módot kliens-szerver modellnek is nevezni, ahol az elnevezés arra utal, hogy minden modul szolgáltatásokat biztosít az ezt igénylők számára. Az ilyen felépítésű rendszerek javítása-módosítása egyszerűbb és főként biztonságosabb. Ha a rendszer szabályait betartjuk, akkor egy módosításnak sokkal kevésbé lehet tovagyűrűző hatása, mint ahogy az a monolitikus rendszereknél előfordul. A modulhívások, a definiált paraméterátadási szabályok precíz betartása a rendszer működési sebességében jelenthet némi terhet. Ilyen moduláris felépítésű rendszerek például a Linux-rendszerek.

A moduláris rendszerek egy-egy modulja önmagában persze monolitikus is lehet. Ha a modul kicsi akkor ez természetes is, minél nagyobb egy modul annál inkább okozhat ez gondot. Elméleti meggondolásokkal nem lehet eldönteni mi a még elfogadható modulméret*.

így a rendszerek moduláris/monolitikus osztályozása nem mindig egyértelmű. E csoportba sorolhatók a Windows-rendszerek, melyek modulárisak ugyan, de monolitikus jellegű részeik is vannak.

* A strukturált programozási „iskola" például maximum 180 soros modulméretet engedett meg. Elméletileg ez sem megalapozható szám; onnan származik, hogy a mainframe gépeken ez éppen 3 nyomtatott oldal mérete volt, amit még áttekinthetőnek tartottak – sokan vitatják jogosságát. Egy bonyolult feladatot ilyenkor igen sok kis részre kellene bontani, ami meglehet, hogy nem célszerű.

# *2. A MEGSZAKÍTÁSRENDSZER*

Amint a bevezetőben már vázoltuk, a megszakítási rendszer jelentősége abban áll, hogy lehetővé teszi az operációs rendszer működését. Ez a mechanizmus biztosítja azt, hogy ha akár az éppen futó program, akár a perifériák valamelyike az operációs rendszer szolgáltatását, döntését, segítségét igényli, vagy közölni akar az operációs rendszerrel valamit, akkor az operációs rendszer megfelelő rutinja elindul. Azt, hogy az operációs rendszer felügyelete alá tartozó tevékenységeket „közönséges" programok „önhatalmúan" ne végezhessenek (ilyen a legtöbb I/O-tevékenység, a memóriaterületek lefoglalása és felszabadítása) a privilegizált utasítások bevezetésével oldották meg. Amikor a programban ilyen funkciókra van szükség, akkor a program is megszakítást generál, egy megszakítást eredményező utasítás végrehajtásával. Ezzel a megszakítással jelzi az operációs rendszernek, hogy valamilyen igénye van, amit – ha mással nem ütközik – az operációs rendszer kielégít.

## 2.1. A megszakítások forrása, okok szerinti osztályozása

Egy fejlett megszakításrendszerben a megszakítási okok a következőképpen osztályozhatók [23]:

1. *Géphibák.* Ebbe az osztályba tartoznak az automatikus hibafigyelő áramkörök jelzései. A hibafigyelő áramkörök egy része általában hibajelző kódok alkalmazásával (pl. paritáskontrollal) tárja fel az adatátviteli vonalak, CPU-regiszterek, vagy a memória hibáit. A hibafigyelő áramkörök másik része az energiaellátás és a hűtés zavarai esetén jelez.
2. *I/O okok.* Ide tartoznak az I/O-processzorok (csatornák) megszakításkérő jelzései. Ezek általában az I/O-tevékenységek befejeződésekor, vagy adatközlést kezdeményező perifériák bevezető jelzéseikor alakulnak ki.
3. *Külső okok*. Ebbe az osztályba tartoznak a külső eszközök által generált megszakító-jelek. (Ilyen például a nagygépek kezelőpultján a megszakításkérő billentyű lenyomásakor keletkezett jel, vagy összekapcsolt gépek esetén a másik számítógéptől érkező megszakításkérések, az egér billentyűinek vagy a billentyűzet bármely billentyűjének lenyomása stb.) Idetartozik továbbá az intervallum órajele, mely adott időintervallum leteltét jelzi.
4. *Programozási okok.* Az ebbe az osztályba tartozó megszakításkérések mindig valamilyen utasítás végrehajtása, vagy végrehajtásának megkísérlése következményeként alakulnak ki.

A megszakítások célja általában programozási hibák jelzése vagy kiküszöbölése, de a programozó bizonyos célokból szándékosan is előidézhet bizonyos típusú megszakításokat. A programozási okok részletesebb felbontása a következő:

a) *Csapdázott műveleti kódú utasítások alkalmazása.*
A csapdázott utasítások egyik csoportját az extra kódú utasítások alkotják. Ilyenek például az illegális, vagyis a gépi utasításkészletben nem szereplő műveleti kódú utasítások. Ezek véletlen használata programhibát jelent, ezért külön csapdázó hardver figyeli az ilyen kódokat, és végrehajtásukat elnyomva, megszakítást idéz elő. Lehet szándékos is egy illegális utasítás használata, ha a megszakítás-feldolgozó rutint ennek megfelelően írjuk meg. Így például a gépi utasításkészlet mesterségesen kibővíthető problémaorientált utasításokkal, amelyeket a megszakításkezelő rutin hajt végre szubrutinszerűen, majd visszaadja a vezérlést a programnak. Hasonló a helyzet,

ha a hardver egy része opcionális. Például a lebegőpontos aritmetika, vagy a társ- (vagy ko-) processzorok hiánya esetén. Ekkor, ha a programban előfordulnak a koprocesszor utasításkészletébe tartozó utasítások, hardvervégrehajtásuk helyett programmegszakítás történik, és a megszakításfeldolgozó rutin, amely ekkor lényegében szubrutin, szoftvervégrehajtással pótolja a hardvert. A csapdázott utasítások másik csoportját a privilegizált kódú utasítások képezik. Ezeket csak a processzor különleges üzemállapotában (privilegizált állapotnak, vagy védett módnak is szokták nevezni) lehet végrehajtani. A processzor normál üzemállapotában alkalmazva őket, végrehajtásuk helyett megszakítást okoznak. Privilegizált utasítások általában az I/O-tevékenységeket beindító, leállító és ellenőrző utasítások, a memóriavédelem megszervezésével kapcsolatos utasítások, a megszakítás előtti programállapotot visszaállító (visszatérés a megszakítási rutinból) és a diagnosztikai utasítások.

b) *A memóriavédelem megsértése.*
Ha egy program futása során „idegen" (számára tiltott) címre hivatkozik, akkor az alkalmazott hardver-memóriavédelem működésbe lép, és a memóriához fordulás helyett programmegszakítást idéz elő.

c) *A tárkapacitás túlcímzése.*
Ha a konkrét rendszerben a megcímezhető memória kisebb, mint az utasításokban elméletileg használható legnagyobb cím, előfordulhat a tárkapacitás túlcímzése. Ebben az esetben is megszakítás keletkezik.

d) *Címzési előírások megsértése.*
Sok számítógépnél az utasítások operandusai elhelyezésére előírások érvényesek. Ilyenek például, amikor előírják, hogy az operandusok csak páros, esetleg néggyel osztható stb. címeken kezdődhetnek. Ha ezeket az előírásokat nem tartjuk be, akkor az utasítás végrehajtása helyett megszakítás keletkezik.

e) Decimális aritmetikai utasítás operandusában *hibás decimális kódú adat*.

f) *Aritmetikai túlcsordulások és alulcsordulások.*
Általában külön okokként kezelik a fixpontos alulcsordulást, túlcsordulást (az osztó zérus voltát is külön), a lebegőpontos műveletekben a mantissza alulcsordulását, a kitevő lecsordulását, az osztó mantisszájának zérus volta miatti túlcsordulást és a kitevő túlcsordulását.

g) *Az operációs rendszer felügyelőprogramját hívó utasítás használata.*
Ilyenek például az IBM 370, 43xx, 309x gépeken a Supervisor Call azaz SVC, IBM PC gépeken az Interrupt azaz INT-utasítások.

A megszakítások prioritása általában a fenti felsorolás szerinti. Ennek akkor van jelentősége, ha egyszerre, vagy egy megszakítás feldolgozása alatt, egy (több) újabb megszakítás is jelentkezik. Felületesebb osztályozásban szoktak hardver és szoftver, illetve belső és külső megszakításokról is beszélni. Ekkor is a megszakítás kiváltó oka az osztályozás alapja. A „belül" és a „kívül" a processzor szemszögéből nézett helymeghatározás.

## 2.2. A megszakításfeldolgozás hardver része

A megszakítások osztályokba sorolásának prioritási szempontból van jelentősége, a megszakítás hardver- és szoftverfeldolgozása – a hardverhibát jelentő egyes okokat kivéve – egyforma.
Amikor egy megszakítás keletkezik, akkor a processzor az éppen végrehajtás alatt álló utasítást még befejezi. Ezután a hardvernek gondoskodnia kell a megszakított program későbbi folytatásához szükséges információk tárolásáról, a processzor privilegizált állapotba kapcsolásáról, majd a megszakítási rutin elindításáról. Két igen elterjedt géptípuson konkrétan megvizsgáljuk a megszakítás kivitelezését.

*Az IBM 370, 43xx, 390, Z-series processzorok megszakítása*
Ezen a gépcsaládon az éppen futó program (például a megszakítás utáni folytatáshoz is szükséges) fontos jellemzőit az úgynevezett programállapot szóban (ami valójában duplaszó méretű), angol nevén Program Status Word, szokásos rövidítésével PSW-ben tárolják. A PSW egyébként a hardverben együtt elő sem fordul, hanem fizikailag külön kezelt jelzőbitek és bitcsoportok logikai összessége. A szóban forgó gépcsaládnak kétféle üzemmódja, és ehhez kapcsolódóan a PSW-nek kétféle felépítése van. Az egyik a BC (Basic Controll), a másik az EC (Extended Controll) üzemmód. A BC-módban a hardver nem kezel virtuális memóriát, az EC-módban igen. A virtuális memóriával későbbi fejezetben részletesen foglalkozunk.

*A megszakítás eseménye*
A processzor működése közben mindig az aktuális PSW-t használja. A PSW többek között tartalmazza a következőként végrehajtandó utasítás címét is. Mialatt egy utasítás végrehajtása folyik, a hardver mindig megállapítja a következő végrehajtandó utasítás címét. (Ezt soros utasításvégrehajtás során az éppen végrehajtott utasítás címéből és a hosszából – mindkettő a PSW-ből ill. az utasítás kódjából megállapítható – teszi meg, ugró utasítás végrehajtása pedig éppen az új utasításcím beállítását jelenti.) Amikor a processzor egy utasítást végrehajtott, a PSW-ből kiolvassa a következő utasítás címét, az utasítást kiolvassa a memóriából, (az ellenőrzések, pl. érvényes utasításkód, memóriavédelem stb. után, ha mindent rendben talált) megkezdi az utasítás végrehajtását, és így tovább „végtelen ciklusban".

Amikor egy megszakításjel keletkezik, akkor a hardver befejezi az éppen végrehajtás alatt álló utasítást, majd azután megvizsgálja, hogy az aktuális PSW-ben a most jelentkezett megszakítási osztálynak megfelelő maszkbit a megszakítást engedélyezi-e, vagy sem. Ha nem engedélyezi, akkor a megszakítást és ezzel a feldolgozását is várakoztatja, késlelteti. Ha a megszakítás engedélyezett, akkor a jelenlegi programállapot szót („régi PSW") a hardver automatikusan tárolja a megszakítási osztálynak megfelelő „régi PSW" memóriaterületre, és a megszakítási osztálynak megfelelő, a memória rögzített helyén tárolt új PSW kerül kiolvasásra.

A hardver által újonnan betöltött PSW-ben is természetesen van egy „következőutasítás cím", mely éppen az operációs rendszer megszakítási rutinjának a címe. A PSW-csere eredményeként tehát az operációs rendszer megszakítási rutinja indul el. A megszakítási rutin működését befejezve egy LPSW- (load PSW) utasítással valamely korábban tárolt PSW-tartalmat tölt be a PSW „regiszterbe". Ekkor a megszakítási maszkbitek (hiszen ezek is a PSW részei) is új beállításokat kaphatnak, ezt követően a hardver megvizsgálja, hogy van-e várakoztatott további megszakítás, és a jelenlegi maszkok megengedik-e az eddig várakoztatott megszakítást, ha igen, sor kerül erre a megszakításra is, ha nem, akkor ezzel a PSW-cserével valamelyik korábban megszakított program folytatását éri el.

*Az IBM PC megszakítása*
Ebben a gépcsaládban a program működése alatt a megszakított program későbbi folytatásához szükséges információkat az állapotregiszterben, a következőként végrehajtandó utasítás címét pedig külön címszámlálóban tárolják.

Az állapotregiszter (flag) felépítése:

| bit | jelentés |
|---|---|
| 0 | carry – túlcsordulás jelzése |
| 1 | fejlesztésre fenntartva, értéke 0 |
| 2 | parity – eredmény páros volt-e |
| 3 | fejlesztésre fenntartva, értéke 0 |
| 4 | auxiliary carry – aritmetikai túlcsordulás jelzése |
| 5 | fejlesztésre fenntartva, értéke 0 |
| 6 | zero – eredmény 0 volt-e? |
| 7 | signum – előjel, 0 ha az eredmény pozitív |
| 8 | trap – lépésenkénti utasítás-végrehajtás |
| 9 | interrupt – megszakítás letiltva/engedélyezve |
| 10 | direction – ciklusirány (növekvő vagy csökkenő memóriacím) |
| 11 | overflow – túlcsordulás |
| 12–15 | fejlesztésre fenntartva, értékük 0 |

*A memória fix (hardver felosztása)*
A memória elején, a 0000:0000–03FF területen az úgynevezett megszakítási vektortábla található. A táblázat minden eleme 4 bájt méretű, és egy-egy címet (szegmensét és offszetjét) tartalmazza. A memória további felosztása már szoftverfüggő, a gépen működő operációs rendszer memóriakiosztása szerinti.

*A megszakítás eseménye*
A megszakítási igény fellépésekor a processzor megvizsgálja, hogy a megszakítás engedélyezett-e. (Ez programból is befolyásolható, mégpedig az STI – megszakítás engedélyezése – és a CLI – megszakítás tiltása – utasításokkal.) Ha a megszakítás nem engedélyezett, akkor várakozásra kényszerül. Engedélyezett megszakítás esetén veremtárba elmenti az állapotregisztert, az interrupt és a trap flag törlődik, majd a következő utasítás címét (CS- és IP-regiszterek tartalmát) is a veremtárba helyezi, végül a megszakítási számnak megfelelően a memória elején elhelyezett megszakítási vektortábla megfelelő elemét tekinti a meghívandó rutin címének, s erre átadja a vezérlést. (például ha a 17-es megszakítás lép fel, akkor a megszakítási vektortábla 17. eleme, azaz a 0:58–61 memóriacímen talált tartalom tekintődik ugrási címnek.)

A megszakításkezelő rutin, miután elvégezte feladatát, kiadja az IRET-utasítást. Ennek hatására a vezérlés visszakerül a veremtárban található szegmens és ofszet által meghatározott címre (azaz a CS- és IP-regiszterek visszakapják megszakítás előtti tartalmukat) úgy, hogy közben mind az állapotregiszter, mind a veremmutató (stack pointer) visszaáll eredeti értékére. Ha a flag-eket a megszakítási rutin és a program közötti kommunikációra is használjuk, akkor nem az IRET-utasítással lépünk ki a megszakítási rutinból, hanem RET 2-vel. Ezzel elkerüljük az állapotregiszter automatikus visszaállítását.

Ha megszakításfeldolgozás közben újabb megszakítás jelentkezik, akkor az kénytelen kivárni a már folyamatban levő megszakításfeldolgozást.

Az állapotok verembe történő mentése azt jelentené, hogy a megszakítás feldolgozása után az a program fut ismét, melyet a megszakítás megszakított. Az eredeti elgondolás vélhetően ez is volt; így azonban nem volna lehetőség a megszakítás után futtatandó program kiválasztására. Ezt a vermelés nem gátolja, csak körülményesebbé teszi. Ha a későbbiekben vizsgálandó döntési rutin meghatározta, hogy melyik program legyen a futtatott program, akkor a verem megfelelő átrendezésével elérhető, hogy éppen a kívánt program futtatása folytatódjék.

## 2.3. A megszakításfeldolgozás szoftver része

A megszakításfeldolgozás programmal megvalósított (szoftver) része 3 szakaszra bontható. Ezt a *2.1. ábra* szemlélteti.


*2.1. ábra. A megszakításfeldolgozás szakaszai.*

Az engedélyezett, elfogadott megszakítás következtében az éppen futó program végrehajtás alatt lévő utasítása befejeződik, majd – már privilegizált állapotban – a megszakítási rutin indul el. A megszakítási rutin első fázisában kiértékeli a megszakításjelet, és ennek megfelelő feldolgozó rutinnak adja át a vezérlést. A feldolgozórutin elvégzi a megszakításhoz tartozó tevékenységet. A megszakítással kiváltott tevékenység elvégzése után el kell dönteni, hogy a processzor mely programot futtassa tovább (ebből a szempontból egy várakozó megszakítás is futtatandó programnak tekintendő). Ezt a feladatot a döntéshozórutin látja el. Amikor a további tevékenységet a döntéshozórutin meghatározta, akkor a teljes megszakítás lezajlott, a processzor normál állapotba kerül a vezérlés pedig átadódik valamely korábban megszakított (esetleg egy most elinduló) programra.

– Példákkal megvilágítva: felhasználói program futása közben egy korábban elindított mágneslemezes tevékenység befejeződött. E tényről a mágneslemez (vezérlőkön, esetleg csatornákon keresztül) megszakító jelet (és a tevékenysége eredményességére utaló kódot) küld a processzorhoz. Elindulva, a kiértékelő rutin megállapítja, hogy ez olyan megszakítás, amellyel most foglalkozni kell, és azt is, hogy melyik megszakításfeldolgozó rutint kell elindítania. A megszakításfeldolgozó rutin „könyveli", hogy az egyik felhasználói program által korábban elindított mágneslemezes tevékenység – mondjuk eredményesen – befejeződött. Ezt például a programok állapotát leíró táblázatában a várakozási állapot törlésével jelzi. A megszakításhoz szükséges tevékenységet tehát elvégezte. El kell még dönteni, hogy ezek után mely program működjön tovább. E döntést a döntéshozó rutin feladata meghozni. Lehet, hogy az ezen megszakítás által megszakított program fog tovább futni, de lehet, hogy éppen az a program fog tovább futni, amelyik a lemezművelet befejezésére várakozott.

Hasonlóan végiggondolhatjuk, mi történik például akkor, ha egy program működése közben a nyomtatóból kifogy a papír. Ekkor a nyomtató küld egy megszakításjelet a processzorhoz, az éppen futó program megszakad, a megszakítást kiértékelő program eldönti, hogy ezzel a megszakítással foglalkozik-e, s ha igen melyik a hozzá tartozó megszakításfeldolgozó rutin. Az üzenetet küld a gépkezelőnek arról, hogy az adott nyomtatóból kifogyott a papír, majd a döntéshozó programnak adja a vezérlést, s így (esetleg az éppen megszakított) program futhat tovább. Amikor a gépkezelő elhárította a gondot, a nyomtatót üzemképes állapotba hozta (ezt pl. az online gomb megnyomásával jelezte), akkor ez ismét megszakítást eredményez, s a megszakítás feldolgozása után a nyomtatórutin is folytathatja a nyomtatást.

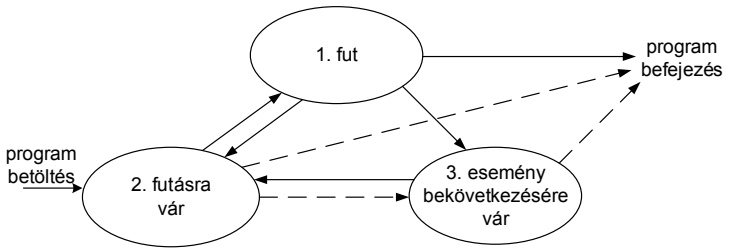
A példák sorát minden lehetséges megszakítási ok bemutatásával folytathatnánk, de bárki önállóan továbbgondolhatja.

A megszakítás most áttekintett, általános, 3 fázisra bontott feladatai közül konkrét gépeken az első és (ritkábban) a harmadik fázisokat esetleg hardverrel oldják meg. Amint már láttuk például az IBM PC-n annak eldöntése, hogy melyik program a megszakításfeldolgozó, a megszakítás számával történik. A harmadik – döntéshozó – fázis pedig vagy egyszerűen elmarad azzal, hogy a megszakítás feldolgozása után, a veremtechnika miatt, mindig a megszakított program futhat tovább, vagy a 2.2. pont végén tett meggondolásoknak megfelelően „valódi" döntés történhet, majd a verem átrendezése, azért, hogy a döntést ténylegesen érvényesítsük (vagyis az a program működjön majd, amelyiket a döntési algoritmus kiválasztott).

*A programok állapota*

A memóriában elhelyezett (tehát már elindított és még be nem fejeződött) programok az operációs rendszer szempontjából, alapvetően háromféle állapotban lehetnek.

1. A program (valamely) processzoron éppen **fut**.
2. A program **futásra kész** állapotban van, a processzorra (vagy a processzorok valamelyikére) vár.
3. A program valamilyen, esetleg éppen általa, korábban elindított **esemény bekövetkezésére vár**.




*2.2. ábra. A programállapotok és a lehetséges állapotátmenetek.*

Amint a *2.2. ábrán* is látható, a programok indulásukkor a 2-es állapotba kerülnek,

– a 3-as állapotból a 2-es állapotba akkor kerül a program, ha bekövetkezett az az esemény, melyre korábban várakoznia kellett;

– a 2-es állapotból az 1-es állapotba akkor kerül a program, amikor az operációs rendszer (döntési rutinja) futásra kiválasztja;

– az 1-es állapotból a 2-es állapotba akkor kerül a program, ha korábban már futott, de ezt egy megszakítás megszakította és ezt követően az operációs rendszer (döntési rutinja) egy másik programot választott ki futásra;
– az 1-es állapotból a 3-as állapotba akkor kerül a program, ha futása közben (egy megszakítással) kér valamit, elindít valamilyen tevékenységet és a kért/indított események bekövetkezéséig nem tud futni, várakoznia kell.

A 3-as állapotból az 1-es állapotba közvetlenül nem kerülhet program (csak a 2-es állapoton keresztül).

A 2-es állapotból a 3-asba látszólag nem kerülhet program (hiszen ha úgyis várakozik, akkor nem tud kérdést feltenni, vagy bármi más tevékenységet elindítani sem). A valóságban azonban előfordulhat ez az állapotátmenet, például ha a program futási körülményeként előírták, hogy csak éjfél és hajnali 2 óra között futhat. Ekkor ha hajnali 2 előtt (például kisebb prioritása miatt) várakozó, de futásra kész állapotba került, ezalatt elmúlik a hajnali két óra, s ezzel eseményre várakozó állapotba kerül át, nevezetesen arra az eseményre vár, hogy ismét éjfél legyen. Hasonló előfordulhat gépkezelői utasításra (HOLD-állapot), s ekkor az ismét kiválasztható állapotba visszasorolásra (RELEASE) vár.

A program rendesen futással (1-es állapotban) fejezi be működését, nem normális esetekben természetesen a programok futása erőszakosan is befejezhető („kilőjük" a programot), akár 2-es, akár 3-as állapotban is volt.

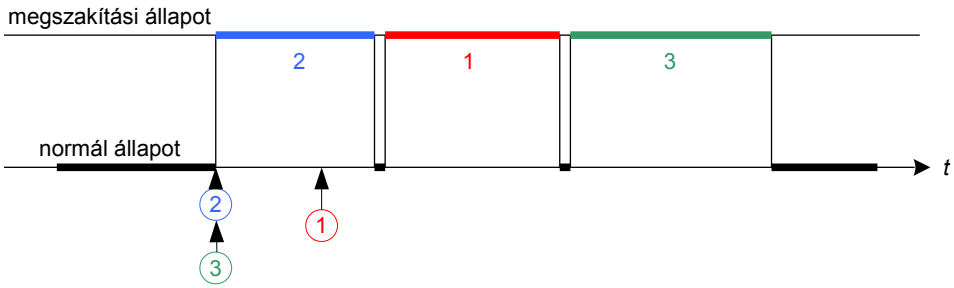
A megszakítás végén működő döntéshozórutin nyilvánvalóan csak a 2-es állapotban lévő programok közül választ.

## 2.4. A megszakításrendszerek típusai

E fejezetben nem valamely konkrét számítógépben megvalósított hardver megszakításfeldolgozást tekintünk át, hanem az operációs rendszerek vizsgálata szempontjából érdekes tipikus hardver-megoldásokat tárgyaljuk.
Aszerint, hogy a megszakításfeldolgozó programok kiszolgálásához hány párhuzamos regiszterkészlete van a processzornak, azaz, hogy a megszakításfeldolgozás közben keletkezett újabb megszakításra hogyan tud reagálni a hardver, beszélünk egy, kettő és többszintű megszakításrendszerről.

### *2.4.1. Az egyszintű megszakításrendszer*

Az egyszintű megszakításrendszer a legegyszerűbb. A normál állapoton kívül egyetlen megszakítási állapotot használ. Ha egy megszakításjel érkezik, és megengedett a megszakítás, akkor a megszakítás létrejön. Ha egyszerre több megszakításjel érkezett, akkor a hardver valamilyen „huzalozott" algoritmusa a nagyobb prioritású megszakítást engedi érvényesülni, a másik várakozni kényszerül. Megszakítás feldolgozása közben érkező újabb megszakításjelek várakozni kényszerülnek. A megszakítás feldolgozása végeztével a rendszer a normál állapotba kerül és a hardver a normál utasításvégrehajtás megkezdése előtt megvizsgálja, hogy van-e várakozó megszakítás, ha van, ezek közül a legnagyobb prioritásút érvényesülni engedi. A megszakítások időbeni lefutását a *2.3. ábrán* szemléltethetjük. Tegyük fel, hogy az 1. a legnagyobb prioritást, 2. a kisebb, 3. a legkisebb prioritást jelenti, valamint, hogy a 2. és 3. megszakítások egyszerre, az 1-es megszakítás később jelentkezik.

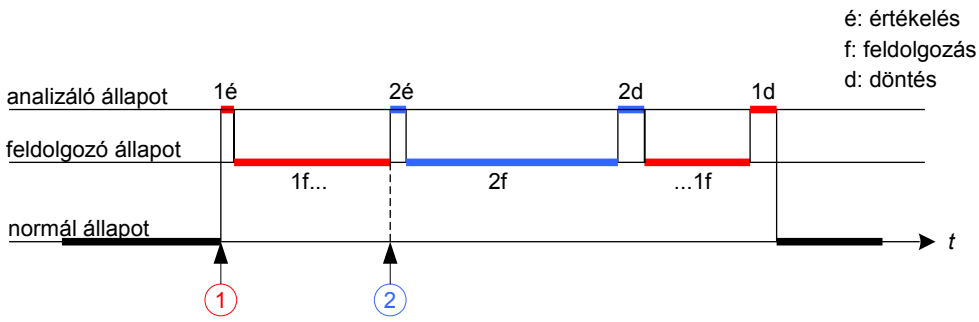



*2.3. ábra. Az egyszintű megszakításrendszer sémája.*

Az egyszintű megszakításrendszer egyszerű, de hátránya, hogy egy kevésbé fontos megszakítás feldolgozása hátráltathatja egy fontosabb(nak tartott) megszakítás feldolgozását, s ezzel a rendszer – felhasználó által érzékelt – teljesítményét rontja.

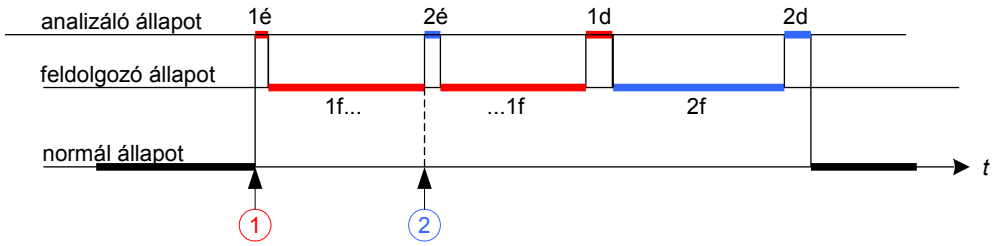
### *2.4.2. A kétszintű megszakításrendszer*

A kétszintű megszakításrendszerben a normál állapoton kívül egy megszakításkiértékelő és egy megszakításfeldolgozó állapotot használnak. Amikor egy megszakító jel érkezik, akkor az (ha nincs kimaszkolva, vagyis az általa jelzett ok szerinti megszakítás létrejöhet) megszakítást okoz, és a megszakítás kiértékelő szintre kerül. A megszakításkiértékelő rutin dönti el, hogy a most érkezett megszakítás feldolgozásába kell-e belekezdeni, vagy ezt várakoztatni kell. Amikor a megszakításfeldolgozás befejeződött, a vezérlés akkor is a megszakításkiértékelő szintre kerül, és ott dől el, hogy melyik várakoztatott megszakításfeldolgozás folytatódjon (vagy kezdődjön el). Ezt a *2.4. ábrán* szemléltetve:




*2.4. ábra. A kétszintű megszakításrendszer sémája – a 2-es megszakítás a nagyobb prioritású.*

A *2.4. ábra* azt a helyzetet szemlélteti, amikor az időben később keletkezett 2. megszakítás nagyobb prioritású, mint a már feldolgozás alatt állt korábban megkezdett 1. megszakítás. Ezért a második megszakítás az első megszakítás feldolgozását is megszakítja. A második megszakítás feldolgozásának végeztével folytatódhat a korábban megszakított első megszakítás feldolgozása. Ha az időben később érkezett 2. megszakítás kisebb prioritású akkor a futás a *2.5. ábra* szerint alakul.

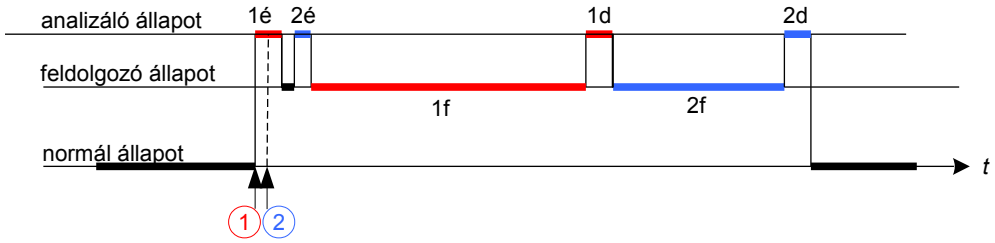



*2.5. ábra. A kétszintű megszakításrendszer sémája – az 1-es megszakítás a nagyobb prioritású.*

Ha a megszakítás kiértékelése közben keletkezik újabb, de kisebb prioritású megszakítás, akkor a helyzet a *2.6. ábrán* látható.

Azaz, amint *2.6. ábra* is szemlélteti, a megszakítás kiértékelése közben az újabb megszakítás várakozásra kényszerül. A folyó kiértékelés befejeztével az érkezett újabb megszakítás kiértékelése kezdődik meg, s annak a megszakításnak a feldolgozására kerül előbb sor, amelyik az értékelés szerint fontosabb.

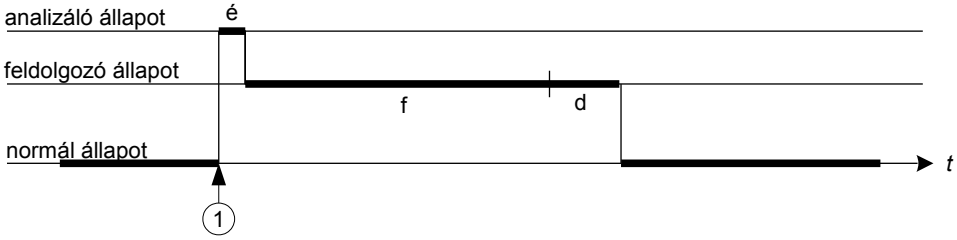
E három ábrából már látszik a kétszintű megszakításrendszer lényege, és hátránya is. A lényege az, hogy a megszakítások kiértékelése gyorsan megtörténik. (Általában azonnal, várakozni csak akkor kell, ha nagyon gyorsan egymás után keletkeznek megszakítások, de ekkor sem a teljes megszakításfeldolgozást kell kivárni, hanem csak a megszakítás értékelését, ami az egész megszakítás kezeléshez képest kicsi idő.)




*2.6. ábra. A kétszintű megszakításrendszer sémája – a 2-es megszakítás az 1-es megszakítás értékelése közben érkezik, az 1-es megszakítás a nagyobb prioritású.*

Ezzel elhárul az egyszintű megszakításrendszer fő hiányossága. A „fontos” megszakítás azonnal, vagy rövid várakozás után érvényesül. E rendszer hátránya viszont az, hogy a fontos megszakítások feldolgozását – igaz viszonylag rövid időkre – megszakítja, ezzel késlelteti azt, az esetleg kevésbé fontos megszakítások kiértékeléséhez szükséges idővel. Ez a módszer tehát nem alkalmazható ott, ahol a nagyon fontos megszakításokra a lehető leggyorsabban kell reagálni.

A kétszintű megszakításrendszer másik esetében a döntési fázis nem az analizáló, hanem a feldolgozó szinten működik, sémáját a *2.7. ábra* szemlélteti.

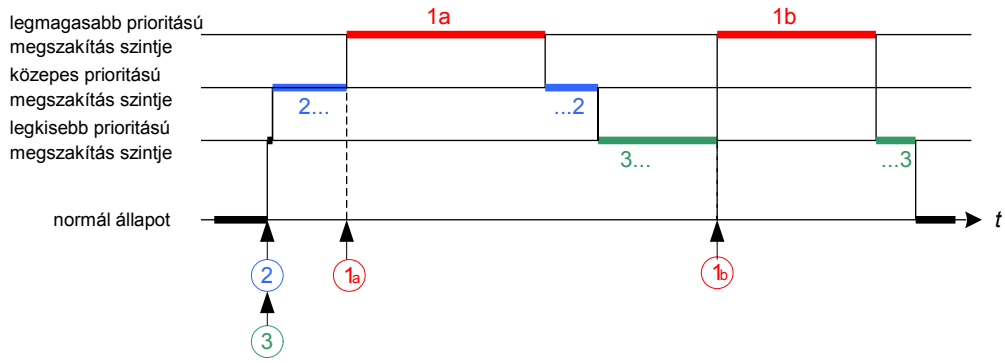



*2.7. ábra. A kétszintű megszakításrendszer sémája – a döntés a megszakító szintre kerül.*

Meg kell itt jegyezni, hogy a „fontos" megszakítás feldolgozásának más megszakítások kiértékelése miatti megszakítása, sok számítógépen a megszakítások maszkolásával elkerülhető. Ha a kimaszkolási lehetőség egy adott géptípuson megvan, akkor is csak igen körültekintően szabad alkalmazni. Nem átgondolt alkalmazása ugyanis megint ahhoz a problémához vezethet, ami az egyszintű megszakításrendszer már elemezett fő gondja. (Nevezetesen, hogy egy általunk fontosnak vélt megszakítás feldolgozása hátráltathatja egy általunk előre nem látott, de adott szituációban még fontosabb megszakítás gyors kiszolgálását.)

### *2.4.3. A többszintű megszakításrendszer*

Többszintű megszakításrendszerrel megoldható az, hogy a fontosabb megszakításfeldolgozás „azonnal" megkezdődjön, és ne zavarhassa semmi, csak egy esetleg annál is fontosabb megszakítás jelentkezése (de ez rendben is van). Elméletileg minden megszakítási okhoz egy-egy önálló megszakítási szintet rendelhetnénk, de ez a hardvert tenné bonyolulttá és nem is indokolt minden megszakítási okot – fontosság szempontjából – különválasztani. A többszintű megszakításrendszerekben a lehetséges megszakítási okokat néhány fontossági osztályba (megszakítási szintre) sorolják. Ha gyakorlatilag egy időben több engedélyezett megszakításkérés érkezik, a kiválasztólogika közülük a legalacsonyabb prioritásút keresi meg, és létrehozza a neki megfelelő szintre a megszakítást. Mindaddig azonban, míg teljesítetlen megszakításkérés maradt, nem engedi a megszakító rutin egyetlen utasítását sem végrehajtani, hanem ismét kiválasztva a még teljesítetlen megszakításkérések közül a legalacsonyabb prioritásút, a neki megfelelő megszakítási szintre visz tovább. Így végülis minden megszakításkérésre azonnal reagál, és a legmagasabb prioritású kérés feldolgozását kezdi meg elsőnek.




*2.8. ábra. A több szintű megszakításrendszer sémája.*

Egy megszakítás feldolgozása után mindig arra a szintre térünk vissza, ahonnan idejöttünk. Az elmondottaknak megfelelően a *2.8. ábrán* bemutatott példában a következő történik: a 2-es és a 3-as osztályokba sorolt okok egyidejű bejelentkezése után megszakítás történik a 3-as ok szintjére, majd onnan azonnal tovább a 2-es ok szintjére. Itt megkezdődik a 2-es megszakítás feldolgozása. Eközben érkező 1-es ok szerinti kérés miatt a feldolgozás abbamarad, és megszakítás történik az 1-es szintre. Itt megtörténik az 1-es megszakítás feldolgozása, majd a vezérlés visszaadódik a 2-es szintre, ahol folytatódik a 2-es megszakítás – előbb megszakított – feldolgozása. Ennek végeztével a vezérlés visszaadódik a 3-as szintre és megkezdődik a feldolgozás. Eközben újabb 1-es szintű megszakítás jelentkezvén, a vezérlés újra az 1-es szintre kerül, és így tovább. Az aktuális megszakításfeldolgozási szintnél magasabb prioritású megszakításkérések várakozás nélkül megszakítást okoznak, a megegyező, vagy alacsonyabb prioritású megszakításkérések várakozni kényszerülnek.

Ezt a megszakításrendszert alkalmazza például az IBM és az ICL a main-frame gépeken.

*A döntési rutin megszakításáról*

Bármelyik megszakításrendszerben, ha a döntési fázis megszakítható, akkor azt később már nem kell folytatni, befejezni. A döntési fázisban az operációs rendszer az akkori helyzetnek megfelelő legjobb döntést igyekszik meghozni. Ha eközben újabb megszakítás következik be, akkor az éppen azt is jelenti, hogy új helyzet alakult ki, az e helyzetben megfelelő döntést ezen megszakítás döntési fázis fogja meghozni, s már semmi értelme a korábbi helyzetnek megfelelő döntés befejezésének.

*A megszakításkezelő rutinok memóriarezidensségéről*

Azért, hogy megszakításkor legyen mit elindítani, az operációs rendszer – legalább e részének – célszerű mindig a memóriában lennie. (Elvileg persze elképzelhető, hogy megszakításkor a hardver töltse be a szükséges programot, de ehhez egyrészt előbb ki kellene mentenie az érintett memóriaterület tartalmát, másrészt ez sok időt venne igénybe, márpedig a megszakításfeldolgozásra a gép működése során sűrűn szükség van, így fontos, hogy minél gyorsabb megoldást használjunk.)

# *3. AZ OPERÁCIÓS RENDSZER ELINDÍTÁSA ÉS LEÁLLÍTÁSA*

A számítógépek bekapcsolásakor azok memóriája üres. (Lehetséges ugyan, hogy „nem felejtő" memóriát használjanak, de ennek használata sok szempontból körülményesebb, rosszul hatna a számítógéprendszer rugalmasságára, így legfeljebb célszámítógépeken és noteszgépeken használatos megoldás.)

Általános célú gépeken a hardvernek lehetőséget kell adni arra, hogy a gép bekapcsolásakor (és bármikor később is) az operációs rendszert (újra) betölthessük és elindíthassuk.

Az operációs rendszert valamilyen adattárolón (manapság többnyire mágneslemezen, SSD-lemezen, pendrive-on vagy CD-n, DVD-n) tartják. Az onnan való betöltését kell tehát a hardverrel megoldanunk. A hardvernek alkalmasnak kell lenni egy program betöltésére és elindítására. Ez valójában nem új feladat, már a régebbi számítógépen is meg kellett oldani. Ott az egyetlen felhasználói program betöltéséről-elindításáról kellett a hardvernek gondoskodnia. Most is hasonló történik, csak a betöltendő program az operációs rendszert betöltő, elindító program lesz. Ennek megoldását két elterjedt architektúrán konkrétan megvizsgáljuk.

## 3.1. IPL az IBM 370, 390 gépcsaládokon

Az operációs rendszer memóriába töltésének első fázisa – az IBM mainframe gépeken – az IPL. (Initial Program Loading, azaz kezdőprogram-betöltés.) Először a számítógépkezelő (operátor) beállítja azon periféria címét, ahonnan az operációs rendszert be kívánja tölteni. Ez rendszerint a mágneslemezegységek valamelyike, de lehet más – program tárolásra alkalmas – periféria is. Ezt követően elindítja az IPL-t. Ekkor a hardver elindít egy Read parancsot a megcímzett I/O-eszközön. Ennek során 24 bájt beolvasódik a memória 0–23. címére. A Read művelet folytatásához a beolvasott 8–15. bájtok első CCW-ként, a 16–23. bájtok második CCW-ként értelmeződnek. További láncolás IPL-nél nem lehetséges. Amikor ez a „csatorna-program" befejeződött, akkor az IPL periféria címe beíródik a memória első szavának (0–3. bájt) 21–31. bitjére, majd a memória 0–7. bájtja új PSW-ként kiolvasásra kerül, az IPL befejeződött, a processzor dolgozni kezd.

Az említett két CCW (Channel Command Word–csatorna parancsszó) összesen 128 Kbájt átvitelét írhatja le, tehát az IPL során maximum 128 Kbájtos programot lehet betölteni. Az így betöltött program még nem az operációs rendszer, hanem egy olyan alkalmas program amely képes az operációs rendszert betölteni, táblázatait inicializálni, fájljait a normál működéshez szükséges állapotba hozni, általában az operációs rendszert üzemképes állapotra hozni.

Példaként nézzünk meg egy egyszerű IPL-programot:

```
IPL1      CSECT
*  CARD-00000001  IPL PSW : 00 00 00 00 00 00 01 E8
*                    CCW1 : 02 0001E0 6 0 00 0050
*                    CCW2 : 02 000228 2 0 00 0050
          ORG   *+400
          USING *-400,15
CARD      DS    CL80                  Ide olvassa be a kártyákat
* CARD-00000002
CCW       DC    X'02000190020000050'  Egy kártyát beolvasó CCW
          LA    15,0                  Bázisregiszternek fogja használni
          LH    12,2(0,0)             Az IPL berendezés címe itt van
          LA    1,CCW
          ST    1,72(0,0)             CAW-ba CCW cím
SIO       SIO   0(12)                 Az IPL címről SIO,  ha jó CC=0
          BC    7,SIO                 CC=1,2,3 esetén  vissza SIO-ra
TIO       TIO   0(12)
          BC    3,TIO                 Müvelet bevárása
          TM    68(0),2               Eszközhiba kérdezés
ALL1      BC    1,ALL1                Itt megáll
          TM    69(0),255             Átviteli hiba kérdezés
ALL2      BC    1,ALL2                Itt megáll
          TM    68(0),4               Teljesülés kérdés
          BC    2,C1
          B     TIO
C1        TM    68(0),1               Egység kivételre kérdés
* CARD-00000003
ALL3      BC    1,ALL3                Itt megáll
          TM    CARD+2,C'X'           Beolvasott kártya 3. byte=X
          BC    1,TXT
          TM    CARD+2,C'N'           Beolvasott kártya 3. byte=N
          BC    14,SIO
END       L     1,CARD+4              END kártyáról indítási cím R1-be
          BR    1                     Ugrás indítási címre
TXT       L     1,CARD+4              Kártyatartalom elhelyezési címe  R1-be
          LH    2,CARD+10             Kártyatartalom hossza R2-be
          BCTR  2,0                   R2-be R2-1  (így L+1 éppen a kívánt)
          STC   2,MVC+1               Hossz az MVC hosszadatra
MVC       MVC   0(56,1),CARD+16       Kártyatartalom átvitele
          B     SIO
          END
```

Amint az a programból látható, ezt IPL-ezve az első 24 bájtban leírt CCW-k betöltik a programot, az IPL PSW kiolvasásával ez el is indul, és alkalmas arra, hogy magas szintű fordítókkal lefordított (azaz ESD-TXT-RLD-END alakú) tetszőleges programot betöltsön és elindítson. Így tudunk viszonylag kényelmesen „önbetöltő", magában működtethető, úgynevezett stand-alone programokat készíteni. Végső soron ilyen az operációs rendszer is.

## 3.2. Bootolás az IBM PC-n

Tekintettel arra, hogy a PC-k felhasználóinak túlnyomó többsége nem számítástechnikai szakember, így különösen fontos tervezési szempont a minél egyszerűbb kezelhetőség, a „felhasználóbarát" kialakítás. Minthogy a PC-t a felhasználó mindig operációs rendszerrel használja, így az operációs rendszer betöltését a gép bekapcsolásakor „automatikusan" elindítják. Természetesen szükség lehet később is rendszer újraindításra/újra betöltésre, ennek megoldása lehetne egy ki-be kapcsolás, ez azonban mint minden elektronikus berendezést a PC-t is nagyon és indokolatlanul megviselné. Így lehetőséget adtak arra, hogy az operációs rendszer betöltését a gép üzemelése közben bármikor a kezelő kezdeményezhesse.

Az IBM (rendszerű) PC-n az operációs rendszer betöltését megfelelő ROM BIOS (rendszerint FFFF:0 címén kezdődő) program indítja. Ez a program vagy a ROM BIOS-ban, vagy a gép setupjában meghatározott sorrend szerint (CD-ről, DVD-ről, „A-drive"–ról, „ZIP-drive"–ról, hálózatról, USB-eszközökről, mint például pendrive-ról, USB-s CD/DVD-ről, stb.) megkísérli az adott adathordozó „logikai elejéről" (mágneslemezszerűen működő perifériák esetén a legelső szektorból) betölteni azt, amit ott talál és programként elindítani. Ha az első perifériáról a betöltés nem sikerül, akkor a következővel próbálkozik. Ha a BIOS-ban felsorolt egyik perifériáról sem sikerülne programot betölteni és elindítani, akkor hibaüzenetet küld a képernyőre. A betöltés címét a ROM BIOS program tartalmazza, szokásos a 0000:7C00 cím. A ROM BIOS betöltő programja ekkor JMP 0000:7C00 ugró utasítással befejeződve, elindítja az általa most betöltött programot. Az így betöltött programot szokták boot („behúzó") programnak nevezni. Ez a program, vagy az általa később betöltődő program a benne meghatározottak szerint gondoskodik az operációs rendszer rutinjainak betöltéséről, a táblázatok, fájlok inicializálásáról, a rendszer üzemkész állapotba hozásáról.

A ROM BIOS által betöltött program floppy esetében a konkrét operációs rendszerhez tartozó boot rekord, azaz tényleges rendszerbetöltést megkezdő, boot program. Merevlemez esetében – annak partíciós szerkezetéhez igazodva – egy „Master Boot Record"-nak nevezett program, amely tartalmazza a lemez felosztását és az „aktív" partíció meghatározását is. Egyszerű esetben a Master Boot Record betöltésével elinduló program az aktív partíció elejéről betölti az operációs rendszerhez tartozó boot (~behúzó) rekordot, s az abban talált program – a már fentebb is leírtak szerint – az operációs rendszert. Ha egy mágneslemez más-más partícióiban más-más operációs rendszereket helyezünk el, akkor az ezek közötti kényelmesebb választást lehetővé tevő programok valamelyikével kicserélhetjük az eredeti Master Boot Record-ot. Az ilyenkor a Master Boot Record-ba kerülő programot a fentiek szerint a bootoláskor a hardver elindítja, s az nem automatikusan az aktív partíciót keresi, hanem a program valami módon (tipikusan menüszerűen) megkérdezi a felhasználót, hogy melyik operációs rendszert kívánja bootolni (elindítani). Számos Linux-csomag lehetővé teszi ilyen bootolási módszer alkalmazását. A Master Boot felülírása kockázatos is lehet, ha itt elrontunk valamit, akkor megtörténhet, hogy a lemezen lévő, egyébként még üzemképes operációs rendszert sem sikerül már bootolni. Másik lehetséges módszer, hogy az aktív partícióban nem egy tényleges üzemi operációs rendszert installálunk, hanem oda kerül a bootmanager program. Ennek a megoldásnak az az előnye, hogy a Master Boot Recordot nem cseréljük le, ezáltal, ha van a lemezen bootolható operációs rendszer akkor annak a partíciójának az aktívvá tételével „direkt" is tudunk operációs rendszert bootolni, hátránya, hogy leköt egy partíciót. Ezt a módszert használja például a PowerQuest cég BootMagic programja. További boot megoldások is szokásosak, gyakori például, hogy az operációs rendszer bootolásakor megjelenítheti a lemezen található más partíciókat (és ha felismeri, vagy korábban

megadtuk számára) az ott található operációs rendszereket, és felkínálhatja azok bootolási lehetőségét is. Így viselkedik a Windows XP, 7, 8, 10 és számos Linux is.

Hálózatban működő PC-k egyik lehetséges indítási módja a **Remoteboot**nak nevezett módszer. Ennek használata azt jelenti, hogy a tekintett PC nem saját perifériá(i egyiké)ről, hanem a hálózaton keresztül valamely más (szerver) gépről kapott programmal indítja a rendszer betöltését. Ez a használatos módszer akkor, ha a tekintett gépet alapvetően csak hálózatban kívánják működtetni, ekkor esetleg nincs is saját mágneslemeze, ez anyagi és nem utolsó sorban program-, adat-, és vírusvédelmi szempontból is nagyon jó megoldás.

A bootolás nemcsak az adott számítógépről kezdeményezhető. Kialakíthatunk olyan rendszert is, amelynek a bekapcsolása és bootolása például modem használatával **távolról is kezdeményezhető.** Helyszíni felügyelet nélküli számítógép alkalmazáskor (de például otthoni gépünk távolról való elindításához is) hasznos lehetőség. Maga a bootolás a már fentebb leírtak szerint történik, csak az indító jel érkezik máshonnan. Ha egy számítógép képes a távolról történő bekacsolásra, akkor ennek lehetőségét és engedélyezését rendszerint a setupban tudjuk szabályozni.

Bootolásra (újra bootolásra) a már működő PC-n több megoldás is van.

Néhány tipikus lehetőség:

– billentyűzeten egyszerre megnyomott ALT-CTRL-DEL* billentyűk hatására a boot program indul,

– ha ez nem „hat" (mert pl. éppen a billentyűzet kezelésében támadt gond) akkor a számítógép házán általában található RESET gombbal is a boot program indítható,

– illetve a futó program is elindíthatja a boot programot.

– Végső megoldásként a számítógép ki- és bekapcsolása is a bootoláshoz vezet.

### 3.3. Az operációs rendszerek leállítása és újraindítása

Az operációs rendszer működésének gyorsabbá tétele végett egy sor információt a memóriában tart. Ezeket valamilyen esemény bekövetkeztekor és/vagy valamilyen időközönként háttér adattárolóra (rendszerint mágneslemezre) is tárolja. Amikor a rendszert normálisan meg akarjuk állítani, akkor az e célra szolgáló parancs kiadása hatására a rendszer elvégzi az állományai aktualizálását, lezárását, és esetleg valóban meg is állítja a rendszert oly módon, hogy az további parancsokat ne fogadjon el (esetleg néhány információkérő, vagy éppen az újraindító parancs kivételével). A rendszer normális leállítása után később elindítva így a megállításkori állapotoknak megfelelő helyzetből folytathatja működését. Ezt „meleg" indításnak is nevezik.

Adódhat olyan eset is (például áramkimaradás, vagy valaminek a meghibásodása, vagy más „durva" ok) amikor a rendszer nem a kezelő utasí-

* A billentyűzeten lenyomott „minden" billentyűkombináció valójában megszakítást jelent, ennek feldolgozó programja határozza meg az adott billentyű (kombináció) hatását. Például WindowsNT-ben a rendszerbe való belépés egy adott fázisában is az ALT-CTRL-DEL billentyűkombinációt kell használni, de az ekkor nem a bootolást indítja. Az OS/2 és a Linux működése közben pedig az ALT-CTRL-DEL billentyűkombináció előbb a rendszer korrekt lezárását kezdeményezi és legfeljebb ezt követően indítja a bootolást. Windows XP működése közben az ALT-CTRL-DEL billentyűkombináció a futó folyamatok listázását teszi lehetővé, sőt beavatkozási lehetőséget is kapunk. Windows 7, 8, 10 rendszerekben az ALT-CTRL-DEL billentyűkombináció hatására egy menü jelenik meg, melyből sokféle tevékenységet választhatunk.

tására áll meg. Ekkor nem tudja a rendes leállításkor elvégzendő teendőit végrehajtani. Ennek következtében a memóriában tárolt adatok, állapotok elvesznek. Újraindulásakor a rendszer ezt a helyzetet észreveheti, és az üzemképesség eléréséhez végrehajthat valamilyen korrekciókat. Többnyire alap (üres) állapotokat tud csak beállítani. Az ilyen indulásokat „hideg" indulásnak nevezzük.

Az operációs rendszerek elindítása és leállítása (a felhasználó szemével nézve improduktív) hosszú idő. Ennek rövidítésére egyes operációs rendszerekben a rendszer **hibernálható**. Ez azt jelenti, hogy az éppen használt memóriatartalmak, processzor- és perifériaállapotok mentésre kerülnek, a rendszer mintegy „lefényképeződik". Az újraindításkor ebből (a többi indítási lehetőségeknél lényegesen gyorsabban) rekonstruálódik a megállításkori állapot, s minden onnan folytatódik, ahol abbahagytuk. Ez gyors módszer ugyan, de azt is jelenti, hogy elmarad a rendszer felépülése, s így csak minden tekintetben változatlan hw/sw környezetben biztosítható a továbbműködés. Ha a hibernálás alatt a számítógépen bármit megváltoztattunk (nemcsak a hardverre vonatkozik ez, hanem pl. ha egyetlen fájl mérete, helye megváltozott), arról a rendszer nem értesül, s ez a továbbműködésben (esetleg csak később érzékelt) zavart okozhat.

Az operációs rendszerek a perifériák túlnyomó többségét meghajtó programok (driver-ek) közreműködésével használják. Mivel nagyon sok különböző periféria létezik, így ezek összességének a meghajtó programját felesleges (és lehetetlen is, hiszen igen gyakran jelennek meg új perifériák) lenne a működő operációs rendszerbe beépíteni. Ezért az operációs rendszerek az általuk működtetett konkrét számítógép perifériái kezeléséhez szükséges meghajtó programokat tartalmazzák. Ha egy új perifériát csatlakoztatunk a számítógépünkhöz (például kicseréljük a videokártyát), akkor az előző konfiguráció működtetésére képes operációs rendszer az új konfigurációt esetleg egyáltalán nem, vagy csak korlátozottan képes működtetni. Ezért az operációs rendszerek elindításakor általában választhatunk valamely (esetleg többféle) **csökkentett módú** indítást. Ilyenkor az operációs rendszer nem használja a perifériák saját meghajtó programját, hanem egy-egy perifériának csak az alapképességeinek tekintett szolgáltatásait veszi igénybe, amiket viszont elvárunk a perifériáktól, hogy mindenképpen képesek legyenek azok biztosítására. (Például a képernyő csak bizonyos felbontásokban, csak kis színválasztékkal működik.) Ezzel az operációs rendszer nem teljes szolgáltatást biztosít ugyan, de lehetővé teszi, hogy az új periféria meghajtó programjára kicseréljük a régit, majd egy újraindítás után az operációs rendszer is, az új periféria is teljes képességeivel, teljes szolgáltatásaival rendesen működhet.

# 4. A PROCESSZORGAZDÁLKODÁS MÓDSZEREI

A processzorral való gazdálkodás azon módszereket jelenti, melyekkel az operációs rendszer (közelebbről a megszakítási rutinok 3., döntéshozó fázisa) eldönti, hogy a futásra váró (2-es állapotú) programok közül a pillanatnyi helyzetben melyik futhat tovább. Az alkalmazott döntési módszerek néhány fő csoportba sorolhatók. A következőkben ezeket tekintjük át.

## 4.1. A prioritásos módszer

A prioritásos módszer lényege az, hogy minden programhoz egy prioritásérték (egy szám) tartozik. A döntéshozó rutin a futásra váró (2-es állapotú) programok közül a legnagyobb prioritásút választja ki, és azt engedi futni. Ha több azonos prioritású program között kell döntenie, akkor többféle eljárás lehetséges.

Az egyik lehetséges módszer szerint a *még legkevesebbet futott* programot engedi tovább működni. Ennek a módszernek az az alapgondolata, hogy ha egy program már úgyis sokáig futott, akkor a felhasználó megelégedettségét nem rontja, ha kicsit még hosszabb idő alatt fut le. Valóban, ha arra gondolunk, hogy egy program 3 óra alatt futhatna le, de így 2 perccel később készül el, ez szinte nem késedelem, ha azonban egy program 1 perc alatt futhatna, akkor a 2 perces késedelem már számottevő, zavaró. Ha egy program még keveset futott, akkor lehet esélye arra, hogy az egy rövid futásidejű program legyen. Ez a módszer az ilyenek számára biztosítja a gyors befejezés lehetőségét.

A másik elterjedt módszer (az előbbivel éppen ellenkezőleg,) azt a programot választja az egyforma prioritásúak közül, amelyik *legrégebben indult*. Így adja meg a lehetőségét annak, hogy az is haladhasson, s végül lefusson. Olyan alkalmazási környezetben, ahol a párhuzamosan futó programok hasonló feladatokat oldanak meg, tehát hasonló a futási idejük, ez az utóbbi módszer biztosítja jobban, hogy minden felhasználás hasonló körülményekkel működhessen.

A programok prioritását rendesen a gépkezelő, a programozó vagy a felhasználó adja meg. Ha ezt elmulasztják, akkor az operációs rendszer rendel valamilyen előre beállított értéket az induló programokhoz. Egyes operációs rendszerek a felhasználói prioritáson („külső prioritás") kívül, úgynevezett belső prioritást is rendelnek a programokhoz. Ezt a program erőforrásigénye szerint valamilyen algoritmussal az operációs rendszer számítja ki, és működés közben dinamikusan módosítja is. Ezzel elérhető, hogy a felhasználók által azonos prioritással elindított programok más-más belső prioritást kapjanak, s ekkor azonos külső prioritás esetén ez a belső prioritás lesz a döntő. A prioritásokat az operátor (gépkezelő) – ha indokoltnak találja – működés közben is módosíthatja.

A prioritásos rendszerekben ügyelni kell arra, hogy kis I/O igényű programok ne kapjanak nagy prioritást, mert ekkor szinte monopolizálhatnák a processzort, és semmilyen más program nem (vagy alig) jutna a processzorhoz, azaz alig vagy sehogy sem haladhatna. A nagy I/O igényű programok nyugodtan kaphatnak nagy prioritást, mert sok I/O műveletük miatt nem képesek a processzor monopolizálására, így az ő várakozási idejeikben más programok is zavartalanul futhatnak.

### 4.2. Az időszeleteléses módszer

A prioritásos módszerrel előfordulhat, hogy egy program monopolizálja a processzort. Ennek a gondnak az elhárítására alkalmas az időosztásos processzorkiosztási stratégia. Az időosztás (vagy időszeletelés, time-slicing, time-sharing) lényege az, hogy a számítógép (egyik) órája bizonyos időközönként megszakításokat generál. Így – mivel természetesen más megszakítások is keletkezhetnek – legkésőbb az órajel megszakításakor a vezérlés az operációs rendszerhez kerül, és ezáltal az eldöntheti, hogy mely program működjön tovább. Ezzel a processzort esetleg monopolizálni akaró programtól a vezérlés elvehető, és más program kaphatja meg azt.

Az operációs rendszerek általában arra is lehetőséget adnak, hogy a programok más-más méretű időszeleteket kapjanak. Az időszeletek nagysága és kiosztása rendszerint működés közben is megváltoztatható. Az operátor (gépkezelő) a terhelés függvényében, vagy más üzemi szempontok szerint rendelkezhet az időszeletelési algoritmus más paraméterezéséről.

Az órajel által kiváltott megszakítás feldolgozásának van egy – minden más megszakítás feldolgozásától eltérő – érdekessége. Ez éppen az, hogy nincs feldolgozási fázisa. Bármely megszakítás pontosan azért következik be, mert valamilyen olyan helyzetet jelez amelyben az operációs rendszernek kell működnie. A megszakítás feldolgozása után (ha további megszakítás már nincs) a döntési fázis által kiválasztott felhasználói program futhat tovább. Az időosztásos rendszerekben az órajellel éppen a döntési rutint kívánjuk működtetni, s ez azt jelenti, hogy ez a megszakítás feldolgozást nem igényel, „egyből" a döntési rutin működhet.

Az időosztásos módszer sok program átlapolt működtetését jól biztosítja. Nem érdemes alkalmazni akkor, ha a gépen (például éjszaka) csak egy, vagy kevés program fut. Egyetlen program futásakor a futást csak hátráltatja, ha órajelek minduntalan megszakítják, és más program éppen nem lévén a döntés csak az lehet, hogy a megszakított egyetlen program futhat tovább.

Első gondolatra érdekes, hogy az időosztásos módszer működhet az órajel-megszakítások nélkül is. Ha ugyanis elegendően sűrűn (az órajelek sűrűségéhez hasonló sűrűn) következnek be megszakítások, akkor azok döntési fázisa működhet az időosztásosnak megfelelő algoritmussal is. Ekkor nem érdemes az órajel-megszakításokkal „hátráltatni" a hasznos munkát. Azt, hogy a rendszerben ehhez elegendően sűrűk-e a megszakítások maga az operációs rendszer mérheti, és az órajel-megszakításokat „kikapcsolhatja", természetesen ha az üzemelési körülmények figyelésével az operációs rendszer ritkának találja a megszakítások jelentkezését automatikusan visszakapcsolhatja az órajel megszakításokat és ezzel a „klasszikus" időosztásos rendszer működik ismét.

Az időosztásos módszer alapesetében a futási jog a futásra váró (2-es állapotú) programok között ciklikusan körbejár.

### 4.3. Kombinált módszerek

A sok felhasználó (és/vagy sok program) kiszolgálására készült operációs rendszerek általában a prioritásos és időszeleteléses módszerek kombinációit alkalmazzák, sok esetben elég kiterjedt paraméterezési lehetőséggel. Az egy program – adott szituációban felhasználható – időszelete lejártakor bekövetkező megszakítás hatására az operációs rendszer kiválasztja a futtatható programok közül az éppen legnagyobb prioritásút, és az kapja a vezérlést. Az ekkor figyelembe vett prioritást részben a felhasználó által megadott prioritás határozza meg, de módosíthatja a gépkezelő, és a rendszer (általában paraméterezhető) algoritmusai is. A módosító algoritmusok a

program eddigi működése során tapasztalt erőforrás-felhasználása alapján növelhetik és csökkenthetik a program kiválasztási prioritását. A rendszer ezen algoritmusának paraméterezése a rendszerprogramozó feladata, és beletartozik a „rendszer hangolása" feladatkörébe. A paraméterek megfelelő beállítására általános szabályokat nem lehet adni, mindig a konkrét alkalmazási területek ismeretében határozhatók meg.

A processzorgazdálkodás konkrét módszerét sok esetben a gépkezelő parancsokkal módosíthatja. A kombinált módszer az időosztás kikapcsolásával a prioritásossá válik, a prioritáskezelés kikapcsolásával pedig az egyszerű időosztásossá változtatható az éppen aktuális gépfelhasználási igényekhez igazodva.

# 5. A MEMÓRIAGAZDÁLKODÁS MÓDSZEREI

A memóriát – az eredeti koncepció szerint – a programok és az adatok tárolására használjuk. Már láttuk, hogy az operációs rendszer (egy részének) is a memóriában kell lennie, a rendszer betöltése és a megszakításrendszer működtetése is a memória speciális felhasználását igényli. Több más célra is célszerű – esetenként szükséges is – a memória egy részét felhasználni. A PC-k esetében például az eredeti elgondolás szerint a képernyőn megjelenítendő kép elhelyezésére, az IBM 43xx gépekben pedig a mikroprogram elhelyezésére is a memóriát használják. A PC-knél később a megjelenítés minősége és gyorsasága különös hangsúlyt kapott, ennek megvalósítását sok esetben úgy oldják meg, hogy a megjelenítést vezérlő videokártyán (mely persze lehet, hogy nem önálló kártya, hanem például az alaplappal egybeépített) önálló processzor és memória is van. Ebben az esetben a képmegjelenítési feladathoz nem feltétlenül kell a „fő" memóriából területet elhasználni. Szintén a memória egy részének felhasználásával oldanak meg periféria-vezérlő feladatokat is. Ilyen például a ROM BIOS a PC-ken, és a „belső terminálvezérlés" az IBM 43xx gépcsaládban. Az operációs rendszerek vizsgálata szempontjából a memória ilyen fix célokra felhasznált területeit az adott számítógép hardver-adottságának kell tekintenünk.

Az operációs rendszer a memória szabadon maradt részeivel gazdálkodhat. E fejezetben a memóriagazdálkodás algoritmusait fogjuk áttekinteni.

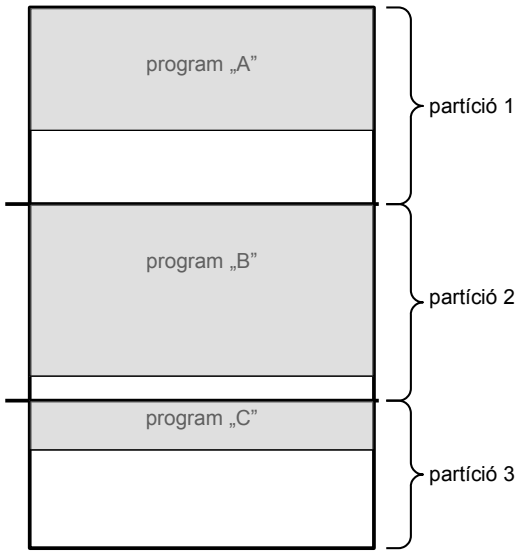
Az operációs rendszer a memória szabad területén helyezkedik el. Vannak bizonyos programjai, amelyeknek feltétlenül a memóriában kell lennie. Más programok csak akkor töltődnek be, amikor éppen szükség van rájuk. Ha egy adott program állandóan memóriában tartása nélkül az operációs rendszer működőképes ugyan, de igen sokszor van rá szükség, akkor a sokszori betöltés nagyon lerontja a rendszer felhasználó által érzékelt hatásfokát. Célszerű lehet az ilyen programokat a rendszer indulásakor betöltetni a memóriába és ott tartani őket. Ezzel ugyan a rendszer indítása valamivel hosszabb időbe telik, de a működés során ez bőven megtérül. Másrészt az ilyen *rezidens*en betöltött programok csökkentik a memóriában szabadon maradt területet, vagyis a felhasználói programok által használható részt. Túl sok rezidens program végül mégiscsak akadályozhatja a számítógéprendszer használhatóságát. Az olyan programokat melyeket nem tettünk rezidenssé, tranziens programoknak nevezzük. Annak meghatározása, hogy mely programokat tegyük rezidenssé, a konkrét alkalmazásoknak megfelelően, a rendszerprogramozó feladata, és (ez is) a rendszer hangolási tevékenységei közé tartozik.

## 5.1. A memória fix felosztása

A memória fix felosztásáról akkor beszélünk, ha az operációs rendszer a szabad memóriát, logikailag részekre osztva kezeli. A memória felosztása esetleg üzemelés közben, kezelői parancs hatására megváltoztatható, azaz a *fix felosztás* csak az operációs rendszer szemszögéből értendő (vagyis a felosztást az operációs rendszer rutinjai automatikusan nem változtathatják meg). A memória részeit memóriapartícióknak nevezzük. Egy-egy memóriapartícióban egy-egy program futhat.

Természetesen a memóriapartícióban legfeljebb akkora program működhet, amely ott elfér. Így minden memóriapartícióban általában marad szabad hely, másrészt a teljes felhasználható memóriában szabadon maradt helyek összességében esetleg lehetővé tennék újabb program indítását, a fix felosztás mellett ez azonban – külön operátori beavatkozás, a partíciók átméretezése

és esetleg új partíció kialakítása nélkül – nem lehetséges. További korlátja a rögzített memóriafelosztásnak, hogy csak akkora program indítható el, mely legalább a legnagyobb memóriapartícióban elfér. Így az átlagosnál lényegesen nagyobb programok futtatása nehézkes, egyedi operátori beavatkozást igényel.

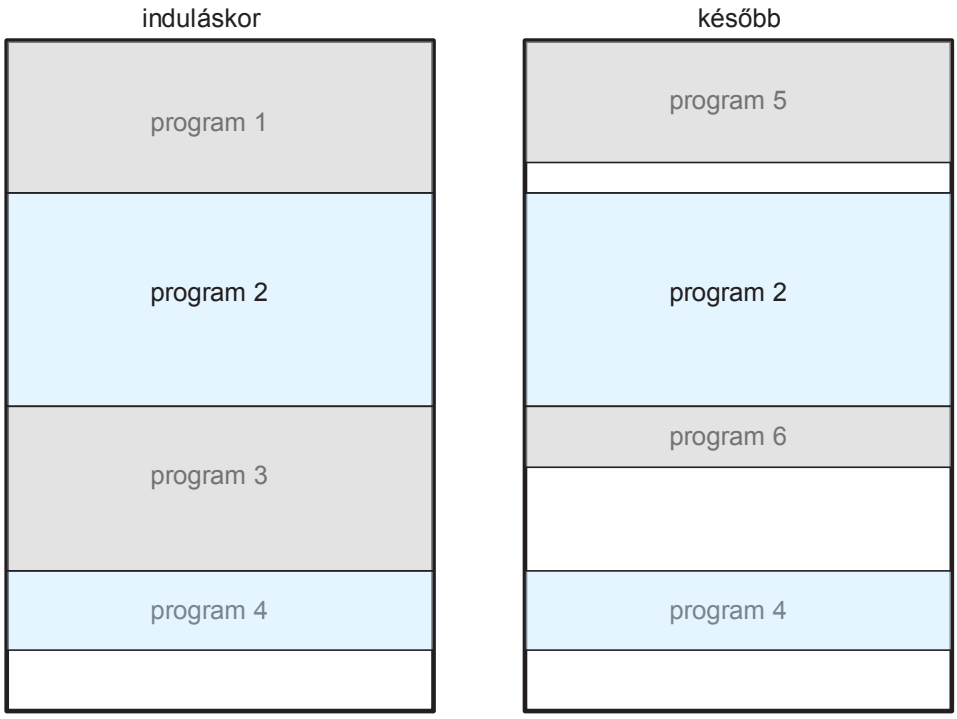



*5.1. ábra. A memória fix felosztása.*

Az operációs rendszer memóriafelosztása üzemelés közben átkonfigurálható, de természetesen csak akkor, ha az átkonfigurálás által érintett összes partíció éppen üres. Ennek elérése-kivárása körülményessé teszi a rendszer használatát. Egyetlen előnye a fix memóriafelosztásnak, hogy viszonylag egyszerű. Az egyszerű feladatokat az operációs rendszer gyorsan és kis rutinokkal tudja megoldani. A memória „szokásos” felosztása a rendszer adott alkalmazási körülményekre történő hangolásának része. Az *5.1 ábrán* a fehéren maradt részek a partíciókban kihasználatlan részek.

## 5.2. A memória dinamikus felhasználása

A memóriafelosztás másik természetes módja a memória dinamikus felhasználása. Ez azt jelenti, hogy minden program akkora memória-területet kap, amekkorát igényel. (A programok memóriaigénye részben automatikusan megállapítható, részben – ha a program önmagában is dinamikusan használja a memóriát, akkor – a program írója vagy kezelője adja meg.) Az operációs rendszer üzemképes állapota elérése után, a programok sorban egymás után a memóriába töltődnek. Ekkor a memória folytonosan töltődik fel. Később, amikor egy program befejeződik, az általa lefoglalt memóriaterület felszabadul. Ha egy következő program a felszabadult helynél kisebb (maximum megegyező) memóriaterületet igényel, akkor ebből azt megkaphatja. Az üzemelés során így a memória sok részén kisebb-nagyobb kihasználatlan hely keletkezik. Az ilyen helyzeteket a programok áthelyezésével oldják meg. Így egyesítve a szabad terület, további programok elindítását teszi lehetővé.




*5.2. ábra. A memória dinamikus felosztása.*

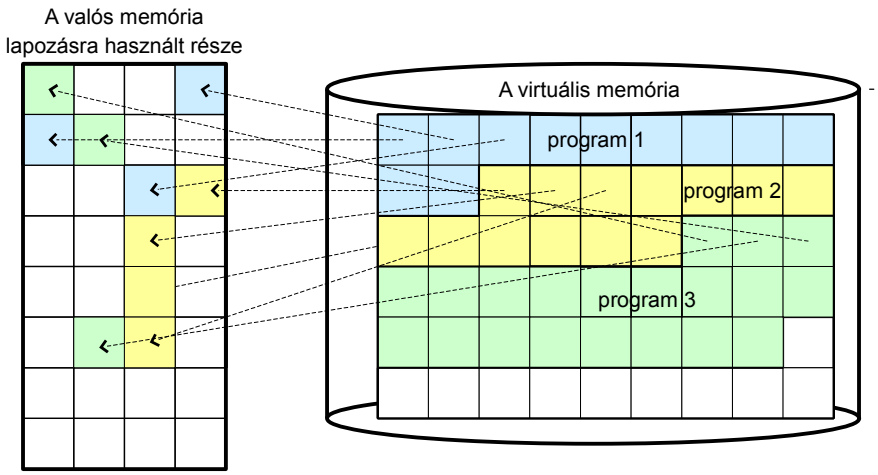
A szabad területek egybegyűjtését garbage collection-nak azaz „szemétgyűjtésnek" is nevezik. Ez ugyan némi időt vesz igénybe, de teljesen automatikusan elvégezhető. Ezáltal a memóriába mindaddig programok tölthetők, amíg a fennmaradt szabad hely arra elegendő. Az átlagosnál nagyobb programok is – ha egyáltalán elférnek a memóriában – külön beavatkozás nélkül futtathatók. A memória dinamikus felosztása természetesen az operációs rendszerre valamivel nagyobb feladatot ró (azaz hasznos helyet és időt köt le), mint a fix felosztás kezelése.

## 5.3. A virtuális memória

A processzorok sebességének jelentős növekedése azt is eredményezi, hogy egyre több program kiszolgálására képesek. Sok program memóriában tartása nagy memóriát igényel. A számítógéppel a felhasználók akkor igazán elégedettek, ha az a csúcsterhelések kezelésére is alkalmas. „Természetesen" lehet a csúcsigényekhez elegendő méretű memóriát építeni-vásárolni, de egyáltalán nem biztos, hogy ez lenne a gazdaságos megoldás, és még így is korlátozná a használható memória méretét, azaz mégsem elégítené ki maradéktalanul a csúcsigényeket.

A majdnem tetszőleges méretű memória biztosítását oly módon oldják meg, hogy háttértárolót (többnyire mágneslemezt) használnak memóriaként. Az ilyen memóriakezelési módszert virtuális memóriakezelésnek nevezik. Az elnevezés arra utal, hogy a programok memóriaként látnak és használnak a valóságban nem létező (látszólagos) memóriaterületeket is. Természetesen a processzor továbbra is csak a valódi memóriában elhelyezett adatokon, a valódi memóriá-

ból kiolvasott műveleteket tudja végrehajtani. Gondoskodni kell tehát arról, hogy az éppen végrehajtásra következő utasítás, és a végrehajtásához szükséges adatok bekerüljenek a valós memóriába. Ennek megoldására mind a valós- mind a virtuális memóriát azonos méretű területekre, lapokra osztják. Az operációs rendszer lapozási-rutin feladata ezután, hogy az éppen szükséges lapokat a virtuális memóriából a valós memóriába betöltsék. Így a valós memória tökéletesen kihasználható (esetleg néhány lap egy-egy részét kivéve).




*5.3. ábra. A virtuális- és a valós memória kapcsolata.*

Egy tetszőleges méretű program utasításai végrehajtásához, ha mind az utasítás, mind az összes (maximum 3) operandusa más-más lapra esik, akkor is elegendő 4 lap a valós memóriából*. Ha egy utasítást vagy adatot tartalmazó lap nincs bent a valós memóriában, de szükség van (vagy várhatóan szükség lesz) rá, akkor az operációs rendszer lapozási-rutin feladata az érintett lap betöltése a virtuális memóriából a valós memóriába.

Az „**igény szerinti lapozás**” (demand paging) módszere csak akkor tölt be lapot a virtuális memóriából a valós memóriába, amikor arra szükség van. Ez garantálja, hogy feleslegesen nem történik belapozás, s ezzel együtt helyfoglalás a valós memóriában, ugyanakkor a programok futásidejét növeli, hiszen gyakran várakozásra kényszerülhetnek a lapozás miatt is.

Másik lehetséges stratégia az „**előretekintő lapozás**” (anticipatory paging), amikor a közeli jövőben várhatóan szükséges lapokat „előre” betöltjük a virtuális memóriából a valós memóriába. Annak megjósolására, hogy mely lapokra lesz szükség, számos algoritmust használnak. A jóslási feladat pontosan nem megoldható (ahhoz sok utasítást előre meg kellene vizsgálni, hogy megállapíthassuk, hogy a program mely adatokat fogja használni, s mely elágazási pontján merre fog elágazni). Nagyon nagy erőfeszítéseket tenni a jóslás javítására nem érdemes, a memória árának csökkenésével egyre nagyobb valós memória áll rendelkezésünkre, s ha abban feleslegesen betöltött lapok miatt átmenetileg le is kötünk területeket ennek költsége sokkal kisebb, mint a programok futási sebességében elérhető javulás-haszna.

* A grafikus processzorok és a mátrix processzorok egyes utasításaira ez nem feltétlenül igaz. Előfordul, hogy ezen utasítások nagyobb tömegű adattal dolgoznak, amelyek esetleg több lapon helyezkednek el.

A lapozási algoritmusok lényegében két szempontból térnek el egymástól. Egyrészt abban, hogy ha már nincs lapozásra használható szabad terület a valós memóriában, akkor mely korábban betöltött lap által elfoglalt területet jelöli ki a most betöltendő lap számára, másrészt, hogy alkalmaz-e valamilyen előretekintő betöltést. A lapozási algoritmusok általában figyelik, hogy egy-egy lap tartalma a betöltése óta megváltozott-e. Ha felszabadításra kijelölt lap tartalma nem változott, akkor azt nem kell a virtuális memóriába visszatölteni, hiszen onnan került be, ott ugyanabban az állapotban megtalálható, ha a tartalma megváltozott, akkor területének szabaddá nyilvánítása előtt „ki kell lapozni" (vissza kell tölteni aktuális tartalmát a virtuális memóriába), hiszen később még szükség lehet rá.

Néhány gyakrabban alkalmazott lapozási algoritmus:

**Legrégebbi lapalgoritmus (FIFO)**: azt a lapot jelöli ki új lap betöltésére, amely a legrégebben került a valós memóriába.

**Újabbesély-algoritmus (second chance)**: a FIFO-algoritmus olyan változata, melyben – ha vannak nem használt lapok is – a nem használt lapok közül a legrégebben valós memóriába került által lefoglalt területet szabadnak nyilvánítja.

**Legkevésbé használt (LFU, Least Frequently Used, vagy NFU, Not Frequently Used) algoritmus**: Abból a feltevésből indul ki, hogy a közelmúltban nem, vagy csak ritkán használt lapokra a jövőben már nem (vagy kis valószínűséggel) lesz még szükség.

Az egyik leghatásosabb lapozási algoritmus az **LRU (Last Recently Used)**, ennek lényegét tekintjük át. Ha a valós memóriában még van üres lap, akkor oda töltődik be a most behozandó lap. Ha a valós memóriában már nincs üres lap, akkor a korábban már virtuális memóriából betöltött és azóta nem változott tartalmú lapok közül a legrégebben használt lap helyébe töltődik be a most betöltendő.

> Azért a nem változott tartalmú lapokat vizsgáljuk először, mert ezek megvannak a virtuális memóriában, vagyis nem kell őket kimásolni a valós memóriából a virtuálisba, mielőtt területüket szabadnak tekintenénk. A legrégebben használt lapot pedig azért keressük, mert arról a legvalószínűbb, hogy még sokáig – esetleg már sohasem – kell használni.

Ha nem találunk nem változott tartalmú lapot, akkor a legrégebben használt lap tartalmát a valós memóriából a virtuális memóriába „kilapozzuk (mentjük)" (hiszen később ismét szükség lehet rá), ezzel a valós memória ezen lapterülete felszabadul, és a most éppen szükséges lap ide betölthető a virtuális memóriából.

> Annak oka, hogy a legrégebben használt lapot vonjuk ki a valós memóriából a programok úgynevezett „haladó jellege". Gyorsan befejeződő programok esetében az általa használt lapterületek gyorsan felszabadulnak. Hosszabb futásidejű programok (tipikusan batch-programoknál fordul elő) esetében a program „haladó jellegén" azt kell érteni, hogy ha a program egy területén már régen áthaladt, vagyis azt a lapot, amelyen a program ezen része (gépi kódja) van, már régóta nem használta, akkor azt nagy valószínűséggel a későbbiekben sem fogja használni, mert az algoritmus későbbi részeihez ért a program. Természetesen előfordulhat, hogy mégis visszatér a program az algoritmus sokkal korábban használt részeihez, s ezért a virtuális memóriában a program összes lapjai megőrződnek, szükség esetén újra belapozhatók a valós memóriába. Ilyen, „távolról" történő visszatérés a programok túlnyomó többségénél igen ritka. (Azért is mert az algoritmusok nem igény-

lik, s azért is mert a távolról való visszatérés a program áttekinthetőségét, kezelhetőségét jelentősen rontja, emiatt ez elkerülendő programozási stílus, az ilyen megoldás csak igen ritkán szükséges ténylegesen.) Így a lapozási algoritmus olyan lapot szabadít fel, melyre nagy valószínűséggel már nem lesz szükség.

Amikor egy korábban kilapozott lapot újra a valós memóriába kell tölteni (belapozás) akkor egyáltalán nem biztos, hogy a valós memória ugyanazon lapjára kerül, mint ahol már korábban előfordult. Látható tehát, hogy a virtuális memória ugyanazon lapja a program működése során más-más valós lapterületekre kerülhet. Így a gépi utasításokban olyan címzési, a hardverben olyan címszámítási módszert kell alkalmazni, mely a bárhol elhelyezett lapok esetében mindig helyes valós címet szolgáltat. Ezt a címzési módszert virtuális címzésnek nevezzük. A virtuális memória megvalósításához tehát a hadvert is ismét bonyolultabbá kell tenni.

A lapozás (lapbetöltés, illetve a lapcsere) a hasznos utasítás-végrehajtást hátráltat(hat)ja, ezért fontos a minél gyorsabb megoldása, emiatt a lapozást hardver módszerekkel is segítik.

Amikor a memóriaigények nem (vagy csak kicsit) haladják meg a valós memóriában rendelkezésre álló memóriaméretet – e szempontból kis terhelésű időszakokban – nem (vagy csak ritkán) szükséges a memórialapozás, azaz ekkor e módszer egyáltalán nem hátráltatja a gép hasznos üzemelését. Amikor a memóriaigény meghaladja a valós memória lehetőségét, akkor „beindul" a lapozási tevékenység, tehát memóriára várakozás miatt egyetlen program sem kényszerül tartós várakozásra. Az operációs rendszer lapozási rutinjai olyan „előrelátó" algoritmusokat is használnak, melyek alkalmasak arra, hogy előre bekészítsék a később nagy valószínűséggel szükséges lapokat a valós memóriába, elérve ezzel azt, hogy amikor szükség lesz rájuk, akkor már rendelkezésre is állnak. Mindezt az előzetes betöltést egyes géptípusokon az autonóm csatorna a processzor működésével párhuzamosan önállóan végrehajtja, azaz a futó programot nem hátráltatja. Azt, hogy mely lapokra lesz nagy valószínűséggel szükség, a programok működésének „haladó jellegéből" (lásd korábbi megjegyzés) lehet „megjósolni". Az alapmegközelítés az, hogy ha a program valamely lapját használjuk éppen, akkor az esetek igen nagy részében a következő lapra lesz ezután szükség. Ennél sokkal ritkább az az eset amikor egy másik „távoli" lapon folytatódik a program tényleges működése. Ennek korrekt tárgyalása komoly algoritmus- és programozáselméleti, valamint valószínűségszámítási meggondolásokat igényel, ezekkel itt részletesen nem foglalkozunk.

**Lapozott memória és nem lapozott memória**. Bizonyos programok nem működtethetők (vagy csak célszerűtlenül) virtuális memória használatával. Más programokra, programmodulokra pedig nagyon gyakran lehet szükség. Ezek lapozása a rendszer összhatékonyságát csökkentené. A virtuális memóriát használó operációs rendszerek lehetővé teszik, hogy ne a teljes „szabad" memóriát használjuk a virtuális memóriakezelés valós memória területeként, hanem osszuk meg e területet, egy részét nem lapozott módon használjuk, a többi részét lapozzuk. Ezzel az ilyen speciális memóriaigényű és a nagyon gyakran használt programok valós memóriában, valós memóriahasználattal működhetnek.

A virtuális memória kezelésének bemutatott „tiszta" megvalósításában tehát az átlagos programok a virtuális memóriát használják memóriaként. Ez azt is jelenti, hogy a program az indulásakor a mágneslemezes fájljából a virtuális memóriába töltődik, ami szintén mágneslemezen helyezkedik el. Innen laponként betöltődik a valós memóriába. Meglehet, hogy teljesen betöltődik

a valós memóriába, és kilapozódás nélkül lefut. Ekkor bizony a módszer előnyeit nem élvezzük, de a hátrányait (lemezről lemezre töltés és laponkénti betöltés = időveszteség) elszenvedjük.

> A számítógépek valós memóriája egyre olcsóbb, egyre nagyobb memóriát építenek a számítógépekbe, következésképpen nem túl nagy terhelésű időszakokban gyakran előfordulhat, hogy a futtatni kívánt program elfér a valós memóriában.

Emiatt gyakran eltérnek a „tiszta" virtuális memóriakezeléstől és a programokat egyből a valós memóriába töltik, s csak akkor kezdődik a lapozás, ha már nem fér el minden éppen igényelt program a valós memóriában. Ekkor viszont természetesen a lapozás algoritmusa is változik, minden a lapozásban érintett lap – tartalmának változásától függetlenül – először kilapozandó a valós memóriából a virtuális memóriába (hiszen még nincs ott).

## 5.4. A VIO, avagy a memóriadiszk

Sok esetben a programoknak a működésükhöz ideiglenes mágneslemez-területekre is szükségük van. Ilyen „munka" területeket használnak a rendező programok az adatfeldolgozó programok (például a valamilyen szempontból különös adatok ideiglenes elhelyezésére, majd további feldolgozás, listázás céljából való visszaolvasására). A programfejlesztésben tipikus fordítás-szerkesztés-futtatás cselekvéssorozatban is szokásos, hogy a fordítóprogram kimenete mágneslemezre kerül, melyet a szerkesztő bemenetként használ (másra többnyire nem is használható), a szerkesztő kimenete ismét lemezre kerül, s onnan a futtatáshoz megint a memóriába kell tölteni. Az ideiglenes és átmeneti lemezhasználat példáit hosszasan folytathatnánk.

Mint minden perifériahasználat, így a lemezhasználat is a processzor és a memóriaelérés sebességéhez képest lassú. Igen sokat segíthet, ha a memóriában jelölünk ki egy (esetleg több) területet, melyet mágneslemezként, az ideiglenes állományok elhelyezésére kívánunk használni. Ennek megoldására az ad jó lehetőséget az operációs rendszernek, hogy a programok önállóan nem használhatnak mágneslemezeket, hanem ha erre van igényük, akkor ennek az igénynek a kiszolgálására egy megszakítással az operációs rendszert kell felkérniük. Ilyenkor az operációs rendszernek jó lehetősége van arra, hogy a tényleges input/outputtevékenység helyett az átviendő adatot egy e célra kijelölt memóriaterületre helyezze el, és később onnan adja vissza amikor a felhasználói program a lemezről történő olvasást kéri. Ezzel elérhető az is, hogy a programban semmiféle átalakítást, vagy az e működéshez való igazítást nem kell tenni. Azt is mondhatjuk, hogy az operációs rendszer a program futási sebessége növelése érdekében „becsapja" a programot. Az ilyen memóriában való mágneslemez-emulációt hardverátalakítás nélkül, tisztán szoftverrel az operációs rendszer meg tudja oldani.

Egyszerűbb megoldásokban, például az MS (PC) DOS memóriadiszk vagy RAMDISK, az e célra fenntartott memóriaterület egy mágneslemeznek látszik, és ha betelik, akkor a programok futásában is zavart okozhat, hiszen a program eredeti szándéka szerint a minden bizonnyal jóval nagyobb mágneslemezt kívánta használni.

Az IBM MVS-ben a memória mágneslemezként való használatát VIO-nak (Virtual Input/Output) nevezik. A VIO mágneslemezként használt memóriaterület betelésekor az operációs rendszer ténylegesen mágneslemezt használ tovább, azaz az történik, amit a program (a programozó) eredetileg is akart. Kis adatmennyiség kezelésében a VIO igen hasznos, jó megoldás, nagy adattömegek kezelésében csak részben segít, de semmit sem ront a programozó eredeti elképzeléséhez képest.

A VIO természetesen csak olyan esetekben használható, ha a fájl tartós tárolására (ténylegesen mágneslemezre írására) nincs igény.

A VIO nem keverendő össze a pendrive (és hasonló nem felejtő memóriák) használatával. Ezek írási/olvasási sebessége lényegesen kisebb a „fő" memória használati sebességénél, sokkal inkább hasonlítható a mágneslemezeken elérhető sebességhez. Így a pendrive-ok használata VIO megvalósításra nem hozna semmi hasznot (adott esetekben akár hátrányos is lehet).

A VIO-megoldás lényege tehát, hogy memóriát mágneslemezként használunk. A már korábban megvizsgált virtuális memória alapgondolata pedig a mágneslemez memóriaként való használata volt. E két módszer látszólag egymás szöges ellentéte. Mindkét megoldás célját és felhasználási területét átgondolva arra a következtetésre kell jutnunk, hogy nem állnak egymással ellentétben. Az IBM MVS operációsrendszer-családban például, mely virtuális memóriát is használ, és módot ad a dinamikus VIO-ra is, egyszerre használják mindkét lehetőséget. Így felgyorsul a mágneslemezes munkaterületeket használó programok futása, és élvezhető a virtuális memória minden előnye is.

## 5.5. A pufferezés

Az adatfájlokkal dolgozó programok számára a fájlművelet alapegysége a rekord. Egy rekord a program egy alap I/O utasításával mozgatható adatmennyiség. A program egy utasításával egy rekordot olvashat be a háttértárolóról, egy rekordot írhat ki, egy rekordot törölhet.

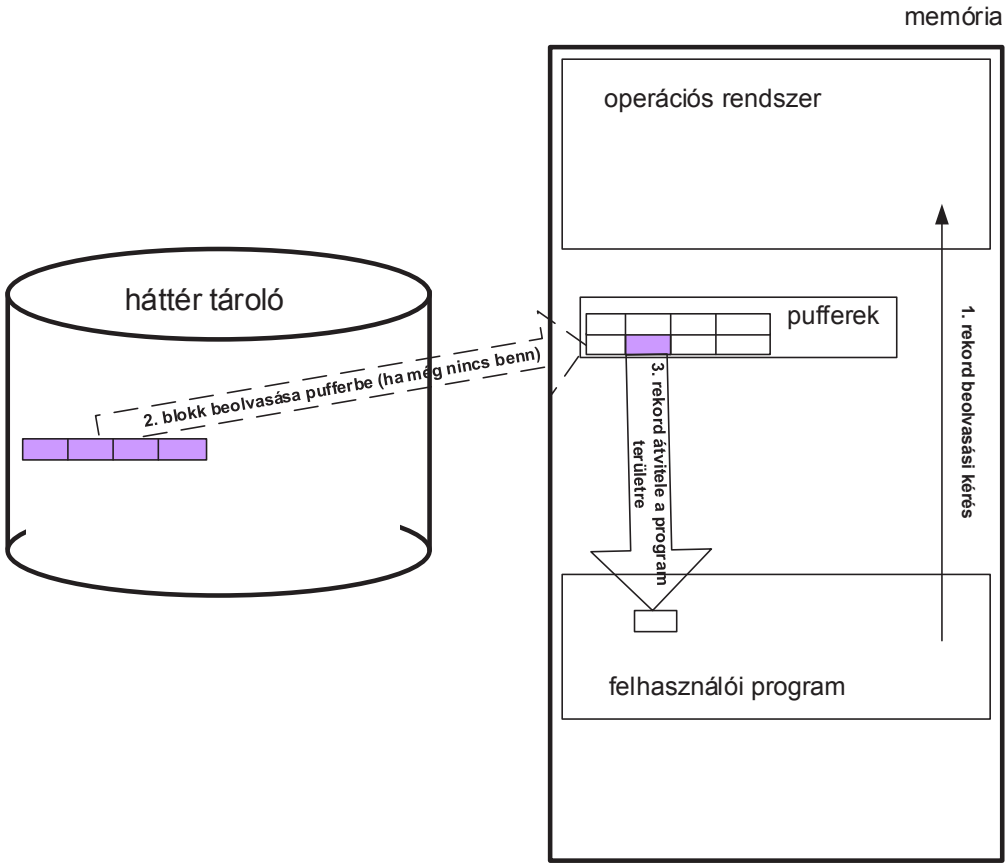
> Egy rekord lehet egy pénzintézeti rendszerben egy számla adatait tartalmazó adattömeg, egy raktári rendszerben a raktári tétel adatait tartalmazó adattömeg, stb. A rekord logikai részei a mezők, egy-egy elemi adategység tárolói, pl. számlaszám, dátum, számlaérték, stb.

A tényleges („fizikai") I/O-átvitelt általában nem célszerű rekordonként végrehajtani. Legrosszabb esetben előfordulhat az is, hogy amikor a program végez egy rekord feldolgozásával, és kéri a következő beolvasását, a beolvasáshoz a mágneslemez (majdnem) teljes körülfordulását ki kell várnia. Ha ez egy nagy (sok rekordból álló) fájl feldolgozásakor minden rekord olvasásánál előfordul, akkor igen sok várakozásra kényszerül a program. Ezen a helyzeten jelentősen segít, ha a fizikai adatmozgatásban nem egy-egy rekordot mozdít meg az operációs rendszer egyszerre, hanem többet. A háttértároló és a memória közötti adatmozgatás egysége (a „fizikai" adatmozgatás egysége) a blokk. Egy blokk általában több (sok) rekord. Az eddigiek értelmében a rekordot a logikai I/O, a blokkot a fizikai I/O egységének tekintjük. A programok minden I/O tevékenységet (megszakítással) az operációs rendszertől kérnek. A program területén egy-egy rekord számára „van hely", a tényleges adatmozgatás az általában lényegesen nagyobb méretű blokkokban történik, így nem lehetséges az, hogy az operációs rendszer „egyből" a program adatterületére hozza be az adatot a háttértárolóról. Minthogy az I/O-tevékenységet az operációs rendszer végzi, módjában áll egy e célra használt memóriaterületre beolvasni a teljes blokkot és abból a program által igényelt rekordot átadni a program adatterületére. Azt a memóriaterületet, ahová az operációs rendszer a fájlok blokkjait beolvassa nevezik **puffer**nek.

A pufferek használatával elérhető, hogy a fizikai I/O az adott adathordozó használatában legcélszerűbb adatmennyiséggel történjen, s erre vonatkozóan ugyanakkor a programokban ne kelljen semmilyen előírást tenni. A pufferek használata általában gyorsítja a fájlokkal foglalkozó

programok működését. Ez különösen a fájlok soros feldolgozásában érzékelhető. Ekkor a program sorban kéri az adatokat, az operációs rendszer beolvas egy blokkot, s abból – már fizikai átvitelre való várakozás nélkül – egymás után több rekordot átad a programnak, s legfeljebb csak akkor kényszerül a program várakozásra, ha újabb blokk beolvasása szükséges.

Ugyanazon fájlhoz több puffer használatával még ez a várakozás is kiküszöbölhető, miközben egyik pufferből kiszolgálja a programot, másik pufferbe bekészítheti a vélhetően következőként igényelt blokkot, s szerencsés esetben (szekvenciális feldolgozásnál biztosan) tényleg ez lesz az a blokk, melyre szükség lesz a program kiszolgálásához. Az éppen használt pufferek cserélgetése miatt ezt a módszert cserepufferezésnek nevezik.




*5.4. ábra. A pufferezés.*

**A blokkméretről.** A blokkméret megállapítása nem az operációs rendszer feladata. Általánosságban az adott adathordozó legjobb használatához igazodó blokkméret választása a célszerű. Figyelembe kell még venni a fájlhasználati módokat. Ha a fájlt tipikusan szekvenciálisan használjuk, akkor minél nagyobb blokkok használata lehet célszerű. Ha a fájlt gyakran véletlen eléréssel is használjuk, akkor a nagy blokkok használata nem szerencsés, hiszen egy-egy rekord eléréséhez ilyen helyzetben „feleslegesen" nagy blokkot kell megmozgatni, ami nagyobb erőforrásigénnyel (memória hely, és átviteli idő) járhat. A blokkméret meghatározásánál a célszerű (alkalmazáshoz és a készülékhez legjobban alkalmazkodó) méretet kell keresnünk.

A PC-k mágneslemezein (floppy és merevlemez) a fizikai adatátvitel mindig szektoronként történik, így itt a blokkolás „klasszikus" módszerének nincs tényleges jelentősége. A blokkolás szerepét később a gyorsítótáras megoldások vették át. Ezeket a fejezet későbbi részében, önálló bekezdésben röviden tárgyaljuk.

**Az aktualizálásról.** A pufferek használata olvasáskor jelentős előnyökkel járhat. A fájlba való írásnál már árnyaltabb a helyzet. A pufferek használatával a program működése közben, amikor a program az operációs rendszertől egy rekord fájlba való írását kéri, ténylegesen „csak" annyi történik, hogy az új (felírandó) rekord a program területéről a pufferbe másolódik át. Ez igen gyorsan történik, viszont még nem kerül az adathordozóra. A program tehát úgy tekinti, hogy egy rekordot már kiírt a fájlba, de ez az írás csak később történik meg. Ha a rekord pufferbe írása után, de a blokk fájlba való írása előtt valamilyen hiba történik, s emiatt a memóriatartalom (ami illékony) elvész, vagy a műveletre más okból nem kerülhet sor, akkor nem kívánt adatvesztés lép fel. Ennek elkerülése, vagy megelőzése érdekében az operációs rendszerek hangolása során általában rendelkezhetünk úgy, hogy bizonyos pufferek tartalma – annak módosulása esetén – haladéktalanul háttértárolóra íródjon, s a program csak ennek eredményes megtörténte után kapja meg azt a visszajelzést, hogy az általa kívánt aktualizálási művelet megtörtént.

Az operációs rendszer a most röviden vizsgált hibalehetőségből származó hiba javítására is felkészülhet. Erre alkalmas például a naplózás. Használatával nem kell a pufferek tartalmát – annak változása esetén – egyből háttértárolóra írni. A műveleteket naplózza a rendszer, s ha hiba lép fel, akkor a napló használatával a helyreállítást el tudja végezni, az adatvesztést kiküszöböli. A késleltetett visszaírásban a legnagyobb megengedett késleltetési idő az operációs rendszer hangolási paraméterei közé tartozhat.

Ha egy operációs rendszer késleltetett visszaírásos stratégiával dolgozik, akkor ez is egyik oka annak, hogy a rendszer leállítása nem történhet drasztikusan (pl. a gép kikapcsolásával), hanem a leállítási utasítás kiadásával, aminek hatására egyebek mellett a még ki nem írt puffertartalmakat is – a leállást megelőzően – a rendszer háttértárolóra írja.

**Mágnesszalagok használata.** A mágnesszalagok használatakor a blokkosítás technikai okból is fontos módszer. (A mágnesszalagos tárolásban csak szekvenciális hozzáférés használata ésszerű, egyes rendszerek nem is tesznek lehetővé más hozzáférési módot.)

A mágnesszalagra való íráskor a szalagot fel kell gyorsítani a normál sebességére, az írás befejeztével pedig le kell fékezni. A gyorsítás-fékezés a szalag egy részének elmozdulásával jár. Ha a szalagra esetleg viszonylag kis rekordonként írnánk fel a fájlt, akkor igen sok helyet használnának el a szalagból a rekordok közötti hézagok (a gyorsítás-fékezés technikai velejárója). A szalag kapacitásának akár 70–80%-a is kihasználhatatlanná válna. Ha blokkosítunk, akkor (a blokk az átvitel fizikai egysége) ezt a kihasználatlan részt drasztikusan tudjuk csökkenteni. Ha pl. 80 bájtos rekordokat egyesével írunk fel, mindegyikhez tartozik majd hézag, ha 200 szoros blokkosítással 16000 bájtos blokkokat használunk a hézagok száma 1/200-ad részére csökken! A szalagon a helykihasználás látványosan javul, a felhasználó által érzékelt átviteli sebesség is nő.

Mozgófejes rendszereknél a szalagokról írottaknak már nincs ekkora, látványos jelentősége, a felhasználó által érzékelt átviteli sebesség persze a blokkosítás-pufferezés hatására ekkor is nő.

Itt a pufferezésnek csak az alapgondolatát tekintettük át. Számos kifinomult pufferezési megoldás létezik, a többszörös pufferek használatának módjai, a pufferek programokból való kezelésének lehetőségei operációs rendszerenként változóak.

**A pufferek számának megállapításáról.** Szerencsére a memória már viszonylag olcsó, és nagy kapacitású. Így eléggé szabadon bánhatunk vele. Ha a konkrét alkalmazási környezetben úgy találjuk, hogy az együttfutó programok maximális száma $p$, a programok (mindegyike külön külön) időben egyszerre átlagosan $f$ számú fájl használnak, az operációs rendszer saját működéséhez $s$ puffert igényel, akkor a rendszer puffereinek számára:

$$\text{alkalmas pufferszám} \geq s + 2 \cdot p \cdot f$$

alsó határérték javasolható.

Ha az adott operációs rendszerben nem a pufferek száma, hanem a pufferek elhelyezésére szánt terület adandó meg, akkor még használnunk kell az átlagos blokkméretet, $b$-t is, így:

$$\text{szükséges pufferterület} \geq (s + 2 \cdot p \cdot f) \cdot b.$$

A formulákban előforduló értékek a konkrét alkalmazási környezet elemzéséből nyerhetők, ebben az operációs rendszer naplózása segítségünkre lehet. Konkrét operációs rendszerek, és egyes más rendszerek (pl. adatbázis rendszerek) a bennük megvalósított adatelérési technikák pontos ismeretében a fenti általános méretezési becsléseknél sokkal pontosabb számítási módokat adhatnak meg a célszerűen használandó pufferek számának és/vagy méretének meghatározására.

**A blokkméret és a pufferek száma együttesen** azt határozzák meg, hogy a fájlból maximum mekkora részt tartunk egyidejűleg a memóriában. Ennek hatására a felhasználói programok úgy érzékelik, hogy a (különösen a szekvenciális) fájlműveletekre fordított idő már viszonylag kis blokkosítás, és néhány puffer alkalmazásával is jelentősen csökken. A blokkméret növekedésével a véletlen eléréshez szükséges fájlelérési idő növekedik (mert esetenként nagy blokkokat kell megmozgatni egy-egy rekord elérése céljából), majd a fájl nagy részének (esetleg az egészének) a memóriába kerülésekor ismét jelentősen csökken a rekord eléréséhez szükséges idő. Nagyon gyakran használt fájlokat – ha van rá lehetőségünk, azaz elegendő a memória a fájl befogadására is – célszerű lehet a memóriában tartani. Sokfelhasználós szolgáltatások (például egyes internet szolgáltatások) esetén a kiszolgálási idő látványosan csökkenthető ezáltal.

**A gyorsítótárakról.** A mai hw/sw-rendszerekben a véletlen elérésre is alkalmas adathordozókon (mágneslemezek, optikai lemezek, különösen az SSD „lemezek" esetében) a blokkosítás-pufferezés fent részletezett módszerének jelentősége csökkent. Szerepét átvette a gyorsítótárak (cache) használata. A gyorsítótár-használat azt jelenti, hogy az adatforgalom mindkét (ki- és beviteli) irányában a pufferekhez hasonló szerepű, de önálló memóriát használnak. Adat beolvasáskor a fájl kért adatot is tartalmazó része (a méret a gyorsítótár méretétől és az adott eszköz hardver- és működési jellemzőitől függ) a gyorsítótárba olvasódik és az operációs rendszer (végül a felhasználói program) adatigényét innen elégíti ki amíg csak lehetséges. A cserepufferezéshez hasonló előolvasási módszereket itt is alkalmazzák. A gyorsítótáras megoldások annyira sikeresek, hogy nem is csak egyszintű, hanem többszintű gyorsítótárazást is használnak. Ekkor egy nagyobb, viszonylag lassabb elérésű és egy általában kisebb, viszont gyorsabb elérésű tárolóból építik fel a gyorsítótáras rendszert. Minél nagyobbak a gyorsítótárak, annál nagyobb esélyünk van arra, hogy az adatigényünk a fizikai mozgásokra való várakozás nélkül kielégíthető legyen.

A gyorsítótárakat nemcsak az adathordozók kezelésében, hanem a processzor és a főmemória közötti adatforgalom felgyorsítására is használják. ($L_1$ és $L_2$ cacheekről szoktak beszélni. $L_1$ = *level 1*, azaz első szintű és $L_2$ = *level 2,* azaz második szintű gyorsítótárak.)

# 6. A PERIFÉRIAGAZDÁLKODÁS

Az előző fejezetben utolsóként tárgyalt témák – a memória mágneslemezként való használata és a pufferezések – átvezetnek az operációs rendszerek erőforrásgazdálkodási feladatai harmadik nagy csoportjába, a perifériákkal való gazdálkodás területére. Ennek vizsgálatához azonban át kell gondolnunk a számítógép-alkalmazás fő módjait is.

## 6.1. A párbeszédes és a kötegelt feldolgozás

A számítógép-alkalmazásnak az igény megfogalmazása és kielégítése közötti idő szempontjából két fő módja van. Az egyik, amikor az igény megfogalmazásával annak kielégítése meg is kezdődik: ez a *párbeszédes* vagy *online* számítógép-használat. Ekkor a felhasználó parancsok, utasítások, menüválasztások formájában közli az igényeit, s a megfelelő program nyomban megkezdi-folytatja működését. A párbeszédes alkalmazásban nem feltétlen kell előre pontosan tudnunk (sokszor nem is tudjuk előre), hogy mely programlehetőségeket, mely adatokkal fogjuk használni. A programmal működése közben is kommunikálunk.

> Gondoljunk például egy pénzintézeti, vagy gyógyszertári, vagy légiforgalmi stb. alkalmazásra. Ekkor meglehet, hogy egész nap egyetlen programot használunk, de nem tudjuk előre, hogy mely részeit és milyen adatokkal; ez ugyanis mindig azon múlik, hogy a következő ügyfelünk, aki éppen hozzánk betéved, mit fog kérni.

A másik szokásos számítógép-alkalmazás, amikor a programfuttatási igény megfogalmazása és a program tényleges működése időben elválik. Azaz magát a futtatási igényt (többnyire még párbeszédes módon) megfogalmazzuk, esetleg a futtatásra vonatkozó körülményeket is előírjuk, de a program futtatására később, további párbeszéd nélkül fog sor kerülni. (Mintegy előre „borítékoljuk" a kéréseinket, igényeinket, a működéshez szükséges paramétereket és adatokat.) Ezt a számítógép-alkalmazási módot *kötegelt*, vagy *batch*-használatnak nevezzük. Jellemző batch-feladat amikor nagy adattömegekkel, előre meghatározható feldolgozást kell végezni, vagy a program hosszú futásidejű. A programmal annak működése közben a felhasználó már nem feltétlenül kommunikál. A kötegelt alkalmazáshoz szükséges feladatmeghatározást *job*nak (munkának) nevezzük.

> Kötegelt alkalmazásra példa lehet egy pénzintézeti kamatszámítás, vagy valamilyen előre megadott szempontok szerinti felszólító levél elkészítése, vagy egy bérlista elkészítése, stb. Ilyenkor előre meg tudjuk adni, hogy mely adathalmazokkal/adatállományokkal/adatbázisokkal (nem külön-külön megadott egyedi adatokkal), milyen feladatot kell elvégezni. Az így összeállított job pedig éppen akkor futhat, ha a párbeszédes ügyfélkiszolgálást már befejeztük, és további szükséges előkészítések is megtörténtek már, például majd éjszaka.

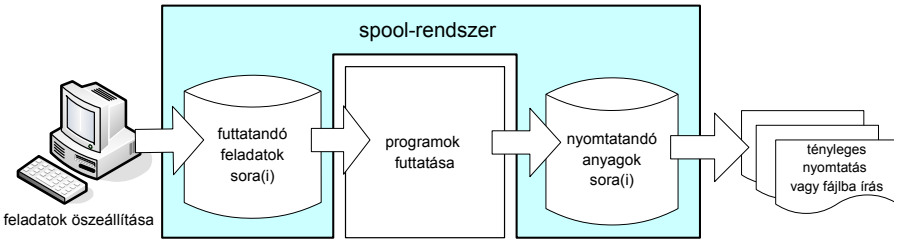
A jobokat megfogalmazásuktól a futtatásukig valahol tárolni kell. Elvileg lehetne ugyan a jobokat a memóriában is tárolni, de órákig (esetleg napokig) a memóriában tárolni olyan adatot, melyre csak majd valamikor lesz szükség, több okból sem célszerű. Pazarlás is lenne, hiszen a memóriaterületekre a lapozáshoz szükségünk van, másrészt például egy ki-bekapcsolás során

elvesznének a feladatok, ez nyilván nem megengedett, valamilyen biztonságos és gazdaságos megoldást kellene találni. A következőként tárgyalandó spool-rendszerek egyik feladata éppen a jobok biztonságos tárolása, kezelése.

* spool: „Simultaneous Pheripheral Operations On-line" rövidítése.

## 6.2. A spool-rendszerek

Amint az előző pontban tárgyaltuk, a batchmunkákat előállításuk és futtatásuk között tárolnia kell az operációs rendszernek. Egészen más okból, de hasonló feladat jelentkezik a nyomtatással kapcsolatban is. Minthogy a nyomtató készülékek számát a napi (esetleg heti) nyomtatási igényekhez célszerű igazítani (némi tartalékkal), így általában nem áll rendelkezésre minden éppen futó programhoz hozzárendelhető nyomtató. Ekkor az az alkalmazott eljárás, hogy a programok futása közben kiadott nyomtató-utasítás tényleges végrehajtása helyett az operációs rendszer a nyomtatásra szánt anyagokat (programonként és listánként) egy-egy fájlban (többnyire mágneslemezen) gyűjti. A tényleges nyomtatást a program futásától eltérő időben a rendszer önállóan hajtja végre. **Az operációs rendszernek a futtatásra várakozó jobok és a nyomtatásra várakozó listák kezelését ellátó részrendszerét spool-rendszernek* nevezzük.** A futásra várakozó jobokat az úgynevezett input queue-ban (input sor), a nyomtatásra várakozó listákat az output queue-ban (output sor) helyezi el.




*6.1. ábra. A spool-rendszer.*

Az input sor(ok) és az output sor(ok) együttesen a spool-területet alkotják. A jobok és listák input és output sorokban való ideiglenes tárolása további praktikus előnyökkel is jár.

– A várakozó sorok természetesen át is rendezhetők, ezáltal a fontosabb jobok vagy nyomtatási anyagok előbb kerülnek feldolgozásra. (A job feldolgozását a futtatása, a nyomtatási anyag feldolgozását a tényleges kinyomtatása jelenti.)

– A nyomtatásnak a futástól való időbeli elválása lehetővé teszi, hogy egy listát tetszőleges példányszámban kinyomtassunk a program megismétlése nélkül. A nyomtatási listában előre-hátra lépkedhetünk, áthidalva az esetleges nyomtatási hibákat (papírbegyűrődés, stb.) a program ismétlése nélkül.

– A hosszú listázás megszakítható és később folytatható, lehetővé válik ezáltal a fontosabb anyagok mielőbbi kinyomtatása.

– A jobok futtatására logikai összefüggéseket adhatunk meg, ezáltal automatizálhatunk bonyolultabb futási rendszereket.

– A feleslegessé vált jobok természetesen törölhetők is. (Erre persze azt is mondhatnánk, hogy minek helyezünk el felesleges jobot, a gyakorlatban azonban lehet, hogy a job-sorozat összeállításakor még nem tudjuk pontosan mire is lesz szükség, és „minden eshetőségre" felkészülünk. Az is meglehet például, hogy délelőtt összeállítunk egy éjszakára szánt futtatást, s később kiderül, hogy arra mégsincs szükség, stb.)

A sorok kezelésének tipikus lehetőségei:
– betekintés a sorok tartalomjegyzékébe,
– betekintés a sorakozó fájlok, jobok tartalmába,
– beszúrás a sorba,
– törlés a sorból,
– a sor átrendezése,
– a sor tagjainak ismétlése,
– a sor tagjainak feldolgozásra való kiválasztása, felfüggesztése, illetve a kiválasztás újra megengedése (hold-release mechanizmus).

A spool-technika egyszerű eszközökkel lehetővé teszi, hogy több számítógépet egységes rendszerben működtessünk. A közös job-sorban esetleg rendelkezhetünk arról, hogy valamely jobot melyik számítógépen kell futtatni, de ha ilyen előírást nem teszünk, akkor az illető job bárhol (ahol előbb lehetősége van) lefuthat. A bármelyik gépen keletkezett lista pedig bármelyik (esetleg fizikailag másik géphez kötött) nyomtatón kinyomtatható. Esetleg arra nézve sincs kikötés, hogy a gépek milyen távolságban lehetnek, akár a szomszéd városban, akár távoli országban is. Az IBM MVS operációs rendszerének JES2 (helyi) és JES3 (távolsági) spool-rendszereivel ilyen többgépes rendszerek alakíthatók ki. Így az egyik számítógépnél dolgozva, összeállíthatunk olyan jobot, illetve „kvázi párbeszédet" folytathatunk olyan programmal, amely a számítógépünkön nem tud futni (mert nincs, esetleg nem is lehet, vagy csak nem érné meg megvenni, mert sem a szoftvert, sem a működtetéséhez szükséges hardvert nem tudnánk kihasználni. Minél nagyobb a számítógép annál inkább előállhat ilyen helyzet).

Fontos megemlíteni, hogy a spool-rendszereknek egyszerűbb, de több gondot okozó megoldásai is léteznek. Jobb híján ezeket ál-spool-rendszereknek nevezhetjük. Ilyen például a MS (PC) DOS nyomtatási technikája. Az ál-spool-rendszerben nem a ténylegesen nyomtatandó anyagot helyezik el a spool-területen, hanem csak egy-egy bejegyzést arra vonatkozólag, hogy mely fájlokat kell majd kinyomtatni. Tudnunk kell, hogy ez azzal járhat, hogy ha a nyomtatási bejegyzést követően, de a tényleges nyomtatás előtt, vagy alatt, az érintett fájlt módosítjuk, töröljük, áthelyezzük, akkor a nyomtatás hibásan történik meg, s erről esetleg még hibaüzenetet sem kapunk, hiszen a rendszer kinyomtatta amit ott talált, vagyis az operációs rendszer szempontjából minden rendben volt. (A felhasználó szempontjából azonban korántsem, mást kapott, mint amit kért.)

A spool-rendszerek egyik leghasznosabb tulajdonsága az lenne, hogy a nyomtatandó anyagokat a tényleges nyomtatás (papír- és festékfelhasználás) előtt megnézhetjük, s ha a nyomtatni szánt anyag nem az eredeti elvárásoknak megfelelő, akkor a tényleges nyomtatásra nem kerül sor. (Idő- és anyagmegtakarítás.) A grafikus nyomtatók igen széles körben való elterjedése ezt az előnyt általánosságban megszüntette. Némi túlzással minden nyomtató más-más (ezért is kell szinte minden nyomtató meghajtó-kiszolgáló programját külön-külön installálnunk). Így olyan program készítése megoldhatatlan, mely bármilyen nyomtatóra szánt anyagot a képernyőn meg tudna mutatni. Néhány nyomtatóhoz adnak olyan programot, amivel ez a feladat (arra az adott nyomtatóra várakozó anyagokra) elvégezhető. Az igényesebb PC-s programok inkább azt a megoldást kínálják fel, hogy maga a felhasználói program ad arra lehetőséget, hogy a nyomtani szánt

anyagokat előre megnézhetjük. (Ilyenek pl. a word, vagy az excel„nyomtatási kép" funkciói.) Ez azonban nem általános, a program készítőjére van bízva, hogy ad-e ilyen jellegű lehetőséget. Az igazán nagy tömegű nyomtatványokat előállító programok (számlázó programok, banki programok) általában sajnos nem biztosítják a nyomtatás előtt a nyomtatvány megtekintésének lehetőségét, az esetleges hiba csak a tényleges nyomtatáskor derül ki, ami nem szerencsés, költségesebb a feltétlen indokoltnál.

A vázolt probléma miatt a PC-s operációs rendszerek spool-rendszere nem teljes körű, azonban ez nem az operációs rendszerek hiányossága, hanem a nyomtatók sokfélesége, és állandónak tekinthető új és új nyomtatók megjelenésének a velejárója. (Sokkal inkább tekinthetjük a szabvány hiányából fakadó problémának, ha lenne egy, vagy néhány, a nyomtatókra vonatkozó szabvány, és azt be is tartanák a gyártók, akkor jó eséllyel lehetne általános nyomtatásianyag-megjelenítő szoftvereket is készíteni. A szabványosításnak jelenleg semmi nyoma, vagy előjele sem tapasztalható. Ezt nem tekinthetjük egyértelműen hiányosságnak, mert így viszont a nyomtatógyártók keze nincs megkötve, szabadon megvalósíthatják bármilyen ötletüket, ezzel elősegítve a technikai haladást.)

## 6.3. Fájlrendszerek

Minden információt, tudást (adatok, programok, szövegek, képek, hangok, filmek) amit a számítógépen kezelünk – igen ritka kivételtől eltekintve – tárolnunk célszerű. A tárolt adatot (programot, szöveget, levelet, képet) később is fel tudjuk használni. Ezért valahogy azonosítani kell őket. A köznapi életben bevált, hogy az azonosítandó dolgoknak nevet (számot, kódot) adunk. Kézenfekvő, hogy a tárolandó adatoknak is nevet adjunk. Egy névvel megnevezett, együtt kezelt adattömeget **fájl**nak (file) nevezzük. Az elméletileg nincs korlátozva, hogy mekkora adattömeg egy fájl. Az alkalmazó tudja eldönteni, hogy mit tekint egy egységesen kezelendő mennyiségnek. Egy fájl lehet egy rövid levél, egy könyv egy fejezete, de maga az egész könyv is, egy fájl tartalmazhatja egy bolti vásárlás adatait, egy napi, vagy akár havi összes vásárlás adatait, egy cégnél dolgozók személyi adatait, stb.

A fájlok kezelésével – tárolás megszervezése, helygazdálkodás, tömörítés, kódolás, stb. – foglalkozó módszereket és az őket megvalósító programrendszert együtt fájlrendszernek nevezik.

A fájlok elnevezésére, méretére minden konkrét fájlrendszerben vannak előírások, korlátozások. A fájl az operációs rendszer szempontjából az egységesen kezelendő adatmennyiség. Az operációs rendszerek adatmanipulációs lehetőségei a fájlokra vonatkoznak. Egy-egy fájlt lehet másolni, törölni, nyomtatni stb. Természetesen a fájlok belső tartalmához is hozzá lehet férni, de az már nem az operációs rendszer lehetősége, hanem futó programokkal tudunk a fájlokon belül adatmanipulációkat végrehajtani. Ilyen programokat rendszerint az operációs rendszerekkel együtt is szállítanak (segédprogramok, utility-k).

Miután a fájlok elnevezésének nyilvánvalóan egyedinek kell lennie, viszont ez nagyban megkötné a számítógép-alkalmazók kezét és vélhetően igen sok zavart, bosszúságot okozna* ezért az operációs rendszerekben általánosan alkalmazott

* Nem használhatna például „read.me" vagy „olvass.el" fájlneveket csak egyetlen szoftver, s ha más gyártó a hasonló célú fájlnak kitalálna akármilyen más nevet, azzal is fennállna a veszély, hogy egy harmadik gyártó is ugyanazt a nevet használná – ez nyilvánvalóan rettenetesen gátolna mindenkit.

megoldás az, hogy a fájlokat csoportokba foglalhatjuk. A fájl csoportokat **könyvtár**aknak (foldereknek, mappáknak) szokták nevezni. A könyvtárak fájlokat és további alkönyvtárakat** tartalmazhatnak. Így többnyire (logikailag) hierarchikus könyvtárszerkezetek alakulnak ki.

Ezzel a fájlok kezelése lényegesen egyszerűsödik. Az elnevezéseknek már csak (al)könyvtárakon belül kell egyedinek lenniük. A fájlokat ezután a helyük, nevezetesen, hogy melyik könyvtár melyik alkönyvtárában találhatóak és nevük alapján azonosítjuk.***

A sok megjegyzésből is érzékelhető, hogy lényegében nem túl sok, de részleteiben igen sokféle fájlrendszert használnak különböző operációs rendszerek. Az operációs rendszerek saját rutinjaikkal néhány fájlrendszer használatára képesek.

További kiegészítő programokkal más (általuk alapértelmezésben nem kezelt) fájlrendszer kezelésére is képesek lehetnek (ez néha teljes körű, néha csak az olvasásra korlátozott).

A konkrét megoldások tehát alapvetően fájlrendszerekhez kötődnek, melyek részben operációs rendszerekhez sorolhatók. A konkrét fájl- és könyvtárrendszerekkel (megoldásaik, előírásaik korlátozásaik) mindig az alkalmazott konkrét esetben kell megismerkednünk.

E jegyzetben nem célom sem kiemelni valamely (néhány) operációs rendszert, sem mindet ismertetni. Az alkalmazható fájlrendszerek megoldásait tekinthetjük technikai paramétereknek, amelyek egy-egy operációs rendszer ismertetéséhez tartoznak.

Még inkább így van ez azon technikai megoldásokra, melyek részleteiben leírják, hogy az adathordozón pontosan hogyan helyezik el az adatokat, milyenek a tartalomjegyzék bejegyzései, a helynyilvántartás módja és még számos részmegoldás.

Hasonlóan az sem tartozik az operációs rendszerek áttekintésébe, hogy a fájlok belső szerkezete milyen, hogy abban hogyan ábrázolnak logikai kapcsolatokat, tulajdonságokat. Ezekkel a kérdésekkel egy-egy alkalmazás ismertetésekor kell foglalkozni. Például azzal, hogy milyen egy .pdf-fájl belső szerkezete, vagy milyenek az adatbázisok fizikai és logikai megoldásai, az ezeket használó szoftverek, és nem az operációs rendszerek foglalkoznak. Az operációs rendszer számára a fájlok belső felépítése indifferens.

A fájlok használata sokszor tudatos, szándékos, az a cél, hogy valamely adatokat későbbi felhasználás céljából megőrizzünk. Ilyenkor az a kifejezett felhasználói szándékunk, hogy fájlokat hozzunk létre.

Máskor is használ az operációs rendszer fájlokat. A kifejezett szándékunk, és ennek megfelelően a végeredmény más, például nyomtatás, de mint láttuk a spool-rendszereknél, azért, hogy a sok „párhuzamosan" futó program nyomtatásai ne keveredjenek, az operációs rendszer a nyomtatási anyagokat átmenetileg fájlokban tárolja, majd később (esetleg) ténylegesen kinyomtatja.

A fájlrendszerek a fájlok és könyvtárak elhelyezésén kívül további szolgáltatásokat is adhatnak. Röviden áttekintve ezeket:

** Nem minden operációs rendszerben lehet a könyvtárnak alkönyvtára, például az IBM MVS-ben sem.

*** Egyes operációs rendszerekben a fájl azonosítása nemcsak a könyvtári helyével és nevével történik, hanem az adathordozó nevét (IBM MVS) vagy a tároló készülék címét (Windows, OS/2) is meg kell adnunk. Többgépes, hálózati környezetben pedig annak a számítógépnek a megnevezése is szükséges lehet, amely perifériáin a fájl megtalálható. Így alakult ki az URL (Uniform Resource Locator), az interneten szabványosított (RFC 1738) egységes erőforrás-azonosító.

**Hozzáférés-ellenőrzés.** Emlékezzünk arra, hogy a multiprogramozott környezetben a felhasználók természetes elvárása, hogy fájljaikhoz illetéktelen programok ne férjenek hozzá. Ezen igény kielégítését a fájlrendszerek segíthetik. Nem csupán a fájlok és könyvtárak elhelyezésével foglalkoznak, hanem ezekhez különböző tulajdonságokat rendelhetnek hozzá. Így feljegyezhetik, hogy az adott fájlt/könyvtárat ki (program, felhasználó) hozta létre, ki mikor módosította, ki mikor olvasta, kinek milyen joggal engedhető meg a hozzáférés stb. A hozzáférés-ellenőrzésre nemcsak a véletlen vagy szándékos illetéktelen fájlhasználat megakadályozása miatt szükséges, hanem az indokoltan és jogosan használt fájlok esetében komoly zavart, tévműködést eredményezhet ha azt egy időben egyszerre több program (vagy ugyanazon program más-más példányai) módosítják. Márpedig a sokfelhasználós rendszerekben ugyanazon adat indokolt, jogos használatára egy időben sok igény fogalmazódhat meg. Elvárjuk, hogy a párhuzamos hozzáférés korrekt megvalósítását a fájlrendszerek is támogassák. Ezt teljes körűen általában csak a magasabb szintű adatkezelő rendszerek (adatbázisrendszerek) biztosítják.

**Hibavédelem.** A fájlok/könyvtárak tárolására használt eszközök túlnyomórészt valamilyen mechanikus működésű készülékek. Ezek természetüknél fogva a használat során kopnak, váratlanul tönkre is mehetnek. Erre fel kell készülnünk. A fájlrendszerek különböző szintű megoldásokat adhatnak/elősegíthetnek a fájlok mentésére/visszaállítására és olyan tárolási módokra, melyek „elviselik" egy vagy több adathordozó tönkremenetelét (pl. RAID-megoldások).

**Tömörítés.** Igen sok fájl – tömörítő algoritmusok segítségével – eredeti méreténél lényegesen kisebb helyen is tárolható. Természetes igény, hogy egy adathordozón a lehető legtöbb fájlt tudjuk elhelyezni. Az adatok hálózaton keresztüli átvitelében is jobb, ha kisebb adatmennyiséget kell mozgatni, ez időben és árban is kedvezőbb. Egyes fájlrendszerek az adatelhelyezéskor tömörítést alkalmaznak, az adat visszanyerésekor pedig helyreállítják a fájl eredeti tartalmát. Így a felhasználói programok nem veszik észre, hogy a tényleges tárolás más bitsorozat formájában történt. Természetesen ilyen tömörítésre csak a *veszteségmentes tömörítések* alkalmasak, melyekből pontosan rekonstruálható a fájl eredeti (nem tömörített) alakja. Számos módszert dolgoztak ki veszteségmentes tömörítésre, ilyenek például a ZIP-, ARJ-, RAR-, TAR-típusú tömörítések.

A kép, mozgókép és hangfájlok már kódoltak, hiszen nem hangot vagy képet tartalmaznak, hanem egy olyan kódsorozatot, amelyet valamilyen hang- vagy képrögzítő eszköz előállított, abból a célból, hogy alkalmas berendezés az eredetihez minél jobban hasonlító hangot vagy képet állíthasson elő. A hang- és képrögzítő eszközök általában valamelyen mintavételezéssel előállt analóg jel digitalizálásával állítják elő az említett kódsorozatot. A mozgókép kódolása pedig – ahogy már a filmeknél kitalálták – állóképek egymásutáni sorozataként valósul meg. Ez digitális kódolás esetén például felkínálja azt a lehetőséget, hogy ha a képsorozat egymás utáni képein majdnem ugyanaz látható, csak egy része változik, „mozog", akkor az egymás utáni képek azonos részét elég egyszer kódolni, s képenként pedig csak a változó részeket kell kódolni-tárolni-továbbítani. Ez máris jelentősen csökkenti a tárolandó adatmennyiséget a képek minőségromlása nélkül.

A kép, mozgókép és hangfájlok tömörítés nélkül igen nagy méretűek is lehetnek, ezért ezeket általában tömörítve szokás tárolni. A kép, mozgókép és hangérzet emberi felhasználás esetén jól rekonstruálható olyan tömörítés alkalmazása segítségével is, amely az eredeti bitsorozat (ami a fenti meggondolások szerint amúgy sem a „valóság", hanem annak már kódolt alakja) előállítására már nem alkalmas. Az ilyen tömörítést *veszteséges tömörítés*nek nevezik. Erre példa az MP3 hangtömörítő, az MPEG2 és MPEG4 képtömörítő eljárások. A veszteséges tömörítések a többi (veszteségmentes) tömörítésekhez képest is jelentősen kisebb méretű fájlokat eredményeznek.

Leegyszerűsítve úgy is fogalmazhatunk, hogy a veszteségért cserébe a kisebb fájlméretet kapjuk.

A veszteséges tömörítő eljárásokban általában paraméterezhetjük a tömörítés mértékét, így nagyon erős tömörítéssel nagyon kis fájlméret érhető ugyan el, de már a kép vagy hangélmény is egyre rosszabb minőségben lesz csak rekonstruálható. Miután az operációs rendszertől nem várható el, hogy „önhatalmúan" eldöntse, hogy mely fájlok és milyen mértékű veszteséggel tömöríthetők, így az operációs rendszerek veszteséges tömörítéssel nem foglalkoznak. A veszteséges tömörítést önálló felhasználói programokkal tudjuk elvégeztetni, az általunk megadott fájlokra, az általunk megadott mértékben. Az eredmény, a tömörített fájl megbízható, veszteségmentes tárolása – mint bármely más fájl esetében is – az operációs rendszer fájlrendszerének a feladata.

**Titkosítás.** A fájlok tartalma általában jelentős érték. Sok esetben igény, néha kötelezettség ezek fokozott védelme. Fel kell készülni például az adatlopásokból (akár önállóan, akár adathordozóval együtt történő lopás, elvesztés, nem kellően gondos selejtezés) fakadó károk elvárható mértékű csökkentésére. Ha a fájl tartalma alkalmasan kódolt-titkosított, akkor a fájl jogtalan használója nehezebben, szerencsés esetben sehogy sem tudja azt ténylegesen felhasználni.

**Visszanyerhetőség/tartalmi törlés.** Egyes fájlrendszerekben a fájl törlésekor csak bejegyződik a fájl attribútumai közé, hogy azt már törölték, de a fájl tartalma még változatlanul elérhető. Ha a töröltséget jelző attribútumot megváltoztatjuk, akkor a fájl ismét elérhető (például a véletlen törlésből visszaállítható). Ha a további munka során szükség van a töröltnek nyilvánított fájl által korábban használt területre, akkor az a terület más fájlok számára kiosztásra kerül, s így a korábban ott tárolt anyag felülíródik.

Nem tudjuk tehát megjósolni, hogy meddig állítható helyre a korábban törölt fájl; lehet, hogy a fájl korábbi helyét másodperceken belül más célra felhasználja a rendszer, de lehet, hogy még napok multán is érintetlen a területe, tehát visszanyerhető. Ha a visszaállíthatóság e bizonytalanságát el akarjuk kerülni, annak egyik lehetősége, hogy a fájlokat törlésükkor egy lomtár könyvtárba másoljuk. Így a fájl valójában nem törlődik, mindaddig, amíg a lomtár könyvtárban tartjuk, onnan visszanyerhető. Természetesen ilyen törlés-stratégia mellett egy fájl törlésével a tárolón hely nem szabadul fel, csak akkor ha már a lomtárból is töröljük, ami rendszerint már tényleges törlést jelent.

Biztonsági, vagy más okokból szükségünk lehet olyan törlésre is, mely után a fájl régi tartalma már biztosan nem nyerhető vissza. Ilyen szolgáltatást egyes fájlrendszerek biztosíthatnak, vagy ennek hiányában az operációs rendszer alkalmas programmal elvégezheti ezt a feladatot. Ekkor nemcsak felszabadul a fájl által korábban elfoglalt lemezterület, hanem annak tartalma is átíródik, így a régi tartalom elvész.

A fájlrendszerek sokféleségének érzékeltetésére, és néhány jellemzőjük összehasonlítására tekintsük át az alábbi (nem teljes!) összefoglalót:
**FAT12** (File Allocation Table, 12 bites címzéssel): A floppy lemezeken használt fájlrendszer.

**FAT16** (File Allocation Table, 16 bites címzéssel): A kisebb mágneslemezeken (max. 2GB/ partíció) használt egyik fájlrendszer. A címzési lehetőségének kicsisége miatt a használni kívánt lemezkapacitás növekedésével a megcímezhető egységek (klaszterek) mérete nő, mivel egy fájl elhelyezésére legalább egy klasztert használ, egyre rosszabb helykihasználást eredményez. (Minden fájl végén az utolsó klasztrer kisebb-nagyobb része üresen marad, különösen rossz lemezkihasználáshoz vezet nagyméretű lemezpartíciókban kisméretű fájlok elhelyezésekor.)

**FAT32** (File Allocation Table, 32 bites címzéssel): A nagyobb (max. 128 GB/partíció) mágneslemezeken használható egyik fájlrendszer. A FAT16 nagy klaszterei helyett sokkal kisebbek használatát teszi lehetővé, jelentősen javít a helykihasználáson.

**HPFS** (High Performance File System, IBM): Eredetileg OS/2-höz kialakított fájlrendszer. Tulajdonságait a *6.1. táblázat*ban foglaltuk össze.

**NTFS** (New Technology File System, MS): Eredetileg a WindowsNT-hez kialakított fájlrendszer. Tulajdonságait a *6.1. táblázat*ban foglaltuk össze.

**VFS** (Virtual File System): Eredetileg a Linuxhoz kialakított fájlrendszer.

**ExtFS, EFS** (Extended File System): A Linuxhoz utóbb kialakított fájlrendszer. Egy francia programozó, Remy Card, alkotta meg. Maximálisan 2 GB méretű partíciók és fájlok létrehozását tette lehetővé és a fájlnevek hossza maximum 255 karakter lehet. Gyenge pontjai: a szabad blokkok és az i-csomópontok kezelése nem bitvektorban, hanem csatolt listában történt. Ez hosszabb működtetés után nagyon felszabdalta a memóriát és így növelte a hozzáférési időt.

**Ext2FS** (Extended File System 2): Az ExtFS továbbfejlesztett változata. Széles körben elterjedt fájlrendszer. Már kiküszöbölték a fragmentációs problémákat és a fájlokra vonatkozó 2 GB-os méretkorlátozást. Ezen kívül az elveszett szektorokat egy speciális könyvtárban (lost+found) tárolja és az esetleges rendszerösszeomlás során megsérült fájlokat az (újra)indításkor felismeri és a javítást egy speciális segédprogrammal (e2fsck) teszi lehetővé.

**Ext3** (Extended File System 3): Az Ext2FS (pl. naplózással) továbbfejlesztett változata.

**ReiserFS** (Hans Reiser által a Linuxhoz kifejlesztett fájlrendszer.) Nagyobb hibatűrő képességű, nagyobb adatbiztonságú fájlrendszer. Egyik érdekessége pl., hogy nincs lehetőség a törölt fájlok visszaállítására.

**JFS** (Journaling File System, IBM). 64 bites és nagyrendszerek kiszolgálására, nagyobb hibatűrő képességű, nagyobb adatbiztonságú fájlrendszer.

**JFS2** A JFS továbbfejlesztett változata.

**ISO 9660** A CD-ken alkalmazott „egyszerű” fájlrendszer.

**Joliet** A CD-ken alkalmazható, az ISO 9660-nál kevesebb kötöttséget megkövetelő fájlrendszer.

**UDF** (Universal Disk Format): Az újraírható CD-ken és DVD-ken alkalmazható egyik fájlrendszer; a CD/DVD mágneslemezhez hasonló használatát teszi lehetővé.

**SFS** (Secure FileSystem): DOS és Windows operációs rendszerekben használható, titkosított, kódolt lemezkötetek kezelését teszi lehetővé.

**NFS** (Network FileSystem): Eredetileg a Unix operációs rendszerekhez kialakított egyik fájlrendszer. A „másik” gépeken tárolt fájlok egyszerű (hálózaton keresztüli) elérését teszi lehetővé.

**UFS** (Unix FileSystem): A Unix operációs rendszerekben használt egyik fájlrendszer.

**WinFS** (Windows Future Storage): A Windows Longhorn változata számára kialakított, az NTFS felváltására készülő fájlrendszer.

**CDFS** (CDROM File System): Az ISO 9660 fájlrendszert kezeli.

**HFS** Macintosh Hierarchical Filesystem.

**ReFS** (Resilient File System): Igen nagy fájlokat is kezelni tudó, önjavító és önszervező fájlrendszer. A Windows Server 2012 R2-ben jelen meg, elérhető a Windows 8, 8.1, 10-ben is.

További fájlrendszerek:

ADFS – Acorn Disc File System
AFFS – Amiga fast filesystem
BeFS – BeOS filesystem
BFS – UnixWare Boot Filesystem CrosStor filesystem
DTFS – Desktop filesystem
EFS – Enhanced filesystem (Linux)
EFS – Extent filesystem (IRIX)
EFS – Encrypting File System (Windows, IBM AIX)
EFS – Elastic File System (Amazon)
FFS – BSD Fast filesystem
GPFS – General Parallel Filesystem
HFS – HP-UX Hi performance filesystem
HTFS – High Throughput FileSystem
LFS – Linux log structured filesystem
MFS – Macintosh filesystem
Minix filesystem
NWFS – Novell NetWare filesystem
NSS – Novell Storage Services
ODS – On Disk Structure filesystem
QNX filesystem
RFS (CD-ROM Filesystem)
RomFS – Rom filesystem
Spiralog filesystem (OpenVMS)
System V and derived filesystems
Text – (Philips' CD-ROM Filesystem)
V7 Filesystem
VxFS – Veritas filesystem (HP-UX, SCO UnixWare, Solaris)
XFS – Extended filesystem (IRIX)
Xia FS

Az operációs rendszerek szempontjából azon fájlrendszerek (egy részét) tekintettük át, melyek használatát az operációs rendszerek lehetővé teszik, felkínálják a felhasználói programok fájljai tárolása céljára. Ettől „*alacsonyabb*", és „*magasabb*" szintű adattárolási szolgáltatásokkal is találkozunk.

Az *alacsonyabb* szint jól érzékelhető a mágneslemezeknél. A mágneslemez hardver/szoftver együttese az operációs rendszer számára bizonyos jellemzőket (fejek száma, cilinderek száma, szektorok száma) mutat, majdnem teljesen függetlenül a tényleges fizikai jellemzőktől. Más esetekben is, az, amit valamilyen konkrét jellemzőkkel bíró mágneslemeznek lát az operációs rendszer, valójában más hardver/szoftver együttes működésének eredménye. Például a RAID-mágneslemezrendszerek, melyek több tényleges mágneslemezt együttesen működtetve nagyobb biztonságú mágneslemez(ek)- ként jelennek meg az operációs rendszer számára. További példák a „pendrive"-nak is nevezett, vagy a digitális fényképezőgépekben is használt flash-memóriák, továbbá az SSD-lemezek, melyek az operációs rendszer számára szintén mágneslemezként mutatják magukat, és az operációs rendszerek fájlrendszereivel használhatók.

A *magasabb* szintű fájlrendszerek olyanok, melyeket nem közvetlenül az operációs rendszer biztosít, hanem az operációs rendszerek fájlrendszereire épülve szolgáltató programok fájlrendszerszerű szolgáltatásokat biztosítanak az ezeket igénylő programok számára. Számos ilyen létezik, például az FTPFS (FTP-fájlrendszer) a VSAM (Virtual Sequential Access Method) és mások.

Ebben a felépítésben adattárolási megoldásként magas szintű tároló szolgáltatásnak tekinthetjük az adatbázisrendszereket is. Ezek is az operációs rendszerek által biztosított fájlrendszerekre épülve, magas szintű adatkezelési lehetőségeket biztosítanak az adatbázisrendszer igénybevevői számára. Részletesebb tanulmányozásuk túlmutat e jegyzet témakörén.

*6.1. táblázat. Néhány fájlrendszer jellemzőinek összehasonlítása.*

| | FAT12 | FAT16 | VFAT | FAT32 | Ext2FS | Ext3FS | ReiserFS | XFS | JFS | NTFS | HPFS | ISO9660 (CD) | Joliet | Mac HFS+ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Legkisebb partícióméret | | 16MB | 16MB | 512MB | | | | | 16MB | | | | | |
| Legnagyobb partícióméret | 16MB | 2GB | 2GB | 128GB | 4TB | 4TB | 17,6TB | | 4PB | 2TB | 512GB | | | |
| Legnagyobb fájlmérete | 16MB | 2GB | 2GB | 4GB | 2GB | 2GB | 4GB | | 512TB | | 2GB | | | |
| Fájlnév (max. char) | 8+3 | 8+3 | 255 | 255 | 255 | | | | | 255 | 255 | 32 | 64 | 255 |
| Max útvonalhossz (char) | 80 | 80 | 260 | 260 | | | | | | 32767 | | | | |
| Változó blokkméret | – | – | – | – | + | + | + | + | + | – | – | – | – | – |
| Defragmentálást igényel–e? | + | + | + | + | – | – | – | + | + | – | – | – | – | – |
| Naplózás | – | – | – | – | – | + | + | + | + | + | + | – | – | – |
| Hozzáférés ellenőrzés | – | – | – | – | + | + | + | + | + | + | + | – | – | + |
| DOS olvas/ír | +/+ | +/+ | +/+ | –/– | –/– | –/– | –/– | –/– | –/– | –/– | –/– | +/+ | +/+ | –/– |
| Windows 9x olvas/ír | +/+ | +/+ | +/+ | +/+ | –/– | –/– | –/– | –/– | –/– | –/– | –/– | +/+ | +/+ | –/– |
| Windows NT olvas/ír | +/+ | +/+ | +/+ | –/– | –/– | –/– | –/– | –/– | –/– | +/+ | +/+ | +/+ | +/+ | –/– |
| Windows 2*, XP olvas/ír | +/+ | +/+ | +/+ | +/+ | –/– | –/– | –/– | –/– | –/– | +/+ | –/– | +/+ | +/+ | –/– |
| OS/2 olvas/ír | +/+ | +/+ | +/+ | –/– | –/– | –/– | –/– | –/– | +/+ | +/+ | +/+ | +/+ | +/+ | –/– |
| Linux olvas/ír | +/+ | +/+ | +/+ | +/+ | +/+ | +/+ | +/+ | +/+ | +/+ | +/– | +/+ | +/+ | +/+ | +/+ |
| Unix olvas/ír | +/+ | +/+ | –/– | –/– | +/+ | +/+ | +/+ | +/+ | +/+ | +/– | +/+ | +/+ | +/+ | –/– |
| MacOS olvas/ír | +/+ | +/+ | –/– | –/– | –/– | –/– | –/– | –/– | –/– | –/– | –/– | +/+ | +/+ | +/+ |

Megjegyzések: az operációs rendszerek és fájlrendszerek kapcsolatában a táblázat az operációs rendszer alapkiépítésbeni lehetőségeit mutatja be. Ezen felül az alapértelmezésben nem kezelt fájlrendszerek olvasásához, vagy íráshoz részben az operációs rendszer készítői, részben tőlük független fejlesztők sok esetben adnak kiegészítő megoldásokat. Ezek használatával az adott operációs rendszer több fájlrendszer kezelésére is alkalmassá tehető.

# 7. TÖBBPROCESSZOROS RENDSZEREK

Az egyprocesszoros számítógépek teljesítménye már oly nagy, hogy jelentősen csak igen drágán fokozható és nem is nagyon nagy mértékben. Számoljunk csak utána, hogy egy (ma már nem is nagy csodálatot kiváltó) mondjuk másodpercenként 100 millió műveletet elvégző processzor egy átlagos utasítás-végrehajtási ideje alatt a fény 3 méter utat tesz meg. Márpedig a fénynél gyorsabban terjedő, olyan fizikai jelenséget, amelyet előállítani és érzékelni is tudunk, nem ismerünk; s így a kapcsolóelemek közötti jelátvitel nem lehet gyorsabb, valamint a kapcsolóelemek műveletvégzésének is ideje van, így – persze egyáltalán nem precízen – végig gondolva beláthatjuk, hogy egyetlen processzor utasítás-végrehajtási teljesítménye már jelentősen nem fokozható.
Ha megnézzük például az Intel processzorokba épített tranzisztorok számát:

| típus | tranzisztor szám (millió db) |
|---|---|
| 286 | 0,134 |
| 386 | 0,275 |
| 486 | 1,2 |
| Pentium4 | 125 |

akkor azt találjuk, hogy a tranzisztorszám (ami hozzávetőlegesen a kapcsolóelemek száma) ezerszeresére nőtt. Ezt a növekedést a processzorral megvalósított utasítások számának növekedése (kb. duplájára) nem indokolja. A növekedés mégis azért következett be, mert a processzoron belül is sok funkciót megsokszoroztak. A processzor már nem egy utasítással foglalkozik egyszerre, hanem több utasítás (különböző végrehajtási fázisokban) feldolgozásával, sőt megjelentek a többmagos processzorok is, amelyek a teljes processzort vagy annak kisebb-nagyobb részének a megtöbbszörözését jelentik egy tokon belül. Mindezek a teljesítmény (az egy időegység alatt végrehajtott utasítások számának) növelését szolgálják.

A teljesítménynövelés természetes (és sok esetben olcsóbb) módja a processzorok számának növelése. A továbbiakban a többprocesszoros rendszereket vizsgáljuk meg az operációs rendszerek szempontjából.

Mielőtt a többprocesszoros rendszerek részletesebb tanulmányozásába fognánk, meg kell említeni az operációs rendszer szempontjából nem, a hétköznapi szóhasználatban mégis több processzoros helyzeteknek tekintett speciális esetcsoportokat.

A számítógépek processzorait jellemezhetjük utasításkészletükkel is. A processzor utasításkészlete a megcélzott felhasználási terület igényeihez igazodik. Az általános célú számítógépeknél a legtipikusabb feladatok megoldásához elegendő utasításkészlettel építik a processzorokat. A konkrét alkalmazások igényelhetnek ennél többet is. A gyártók rendszerint fel is kínálják a hiányzó utasításkészleteket végrehajtani tudó, kiegészítő processzorokat. (Ilyenek az IBM mainframeeknél a decimális és lebegőpontos aritmetika, a mátrixprocesszorok, az IBM-rendszerű PC-knél korábban a ko- vagy társprocesszorok, az overdrive processzorok, a grafikus processzorok.) Ezek a megoldások az operációs rendszer szempontjából nem tekintendők több processzoros rendszernek, hiszen „csak" arról van szó, hogy a gép teljes utasításkészletének egy részét az egyik (külön tokozott) áramköri egységben, más részét egy másik (külön tokozott) áramköri egységben valósítja meg, szerepük normálisan nem felcserélhető. Hasonlóan (ugyan nem utasítás-csoportok, hanem) feladat-csoportok végrehajtására is önálló, (cél) processzorokat alkalmazhatnak,

tehermentesítendő a „fő” processzort. Ilyenek például az I/O vezérlésére, a memória vezérlésére, a lapozásra, a képernyőn megjelenő kép kialakítására és más funkciókra használt processzorok.

A kiegészítő processzorok más vonatkozásban érintik az operációs rendszereket is. Minthogy a kiegészítő processzorok túlnyomó részben opcionálisak – és például a videókártyán gyakran megtalálható processzorok igen sokfélék lehetnek –, így a programozó nem számíthat biztos létezésükre. Ha azonban nem használná ki ezek utasításait a programozás közben, akkor kiegészítő processzorok értelmüket veszítenék. A programozás során tehát kihasználjuk a kiegészítő processzorok lehetőségeit (azaz a programban használjuk ezek utasításait is). Ezzel viszont a számítógépvásárló azt érezné, mintha rákényszerítenék a kiegészítő processzorok megvásárlására is. Az ilyen konfliktusok feloldására is képes az operációs rendszer. Ha a program kiegészítő processzorral felszerelt gépen fut, akkor minden utasítás normálisan végrehajtódik. Ha a gépen nincs kiegészítő processzor, akkor az ennek szóló utasítás végrehajtásának megkísérlése az erre nem képes processzorból „hibás utasításkód” megszakítást vált ki. Mint minden megszakítás hatására, ennek hatására is, az operációs rendszer megfelelő rutinjai kezdenek működni. Ezeket megírhatják úgy is, hogy vizsgálják meg azt, hogy milyen utasítás végrehajtásának megkísérlése váltotta ki a megszakítást, és ha az egy, a kiegészítő processzor által végrehajtható utasítás, akkor modellezze, emulálja ennek működését. Ezzel a módszerrel minden, egyébként helyes program működtethető kiegészítő processzor nélkül is, igaz az utasítás-emuláló rutinok működése miatt lassabban, mint kiegészítő processzor létezése esetén. Ez természetes, megfelel a programozó és az alkalmazó szempontjainak is.

A számítógép „normál”, „fő” processzorát CPU-nak (Central Processing Unit = központi feldolgozó egység) nevezik. A személyi számítógépeknél fokozatosan különösen nagy hangsúlyt kapott a grafikus megjelenítés. Azért, hogy a grafikus megjelenítés feladataival ne kössék le a CPU kapacitásának esetleg jelentős részét is, ezért a grafikus megjelenítést megvalósító elektronikában megjelent önálló processzor és ez a GPU (Graphical Processing Unit) egyre nagyobb teljesítményű. A GPU eredetileg célprocesszorként jelent meg, de későbbi nagy teljesítményének kihasználására az eredeti feladatán túl a mai személyi számítógépekben már, mint ko-, vagy társprocesszort is fel tudjuk használni a programok végrehajtása során.

Nagyon érdekes, részben elméletileg vizsgált, részben már működő megvalósításokban is létező sokprocesszoros rendszerek a sejtautomaták. Ezekben nagyon sok, viszonylag egyszerű, azonos vagy néhány (de nem túl sok) féle processzorból (műveletvégző berendezésből, elemi processzorból itteni szóhasználatban sejtből) álló rendszert építenek. A sejtek rendkívül sokféle kapcsolatban lehetnek a többi sejttel. A működés közben a sejt új állapotát és a kapcsolatain megjelenő kimenő jeleket a sejt korábbi állapota és a bemenő jelek határozzák meg. A szóhasználat is utal arra, hogy a sejtautomatákkal részben a biológiai rendszerek feltételezett működését tekintik példának. Számos tapasztalat szerint ugyanis sok jó megoldás alapötletét adta már a biológiai rendszerek vizsgálata. A sejtautomaták vizsgálata önálló matematikai terület, e jegyzetben nem foglalkozunk vele. Ezt azért is megtehetjük, mert operációs rendszer szempontjából egy ilyen sok processzorból épített processzorrendszer egyetlen processzornak tekintendő. Az elemi sejtprocesszorokat az operációs rendszer nem kezeli, irányítja különállóan.

**Az operációs rendszerek szempontjából azok a (valódi) többprocesszoros rendszerek, melyekben több, logikailag azonos képességű processzorral, az operációs rendszer ténylegesen gazdálkodhat.**

Valódi, egyenrangú többprocesszoros rendszert építhetnek biztonsági meggondolásból is. Ez azt jelenti, hogy ugyanazon utasítást több processzor is végrehajtja, és az eredmény mindig

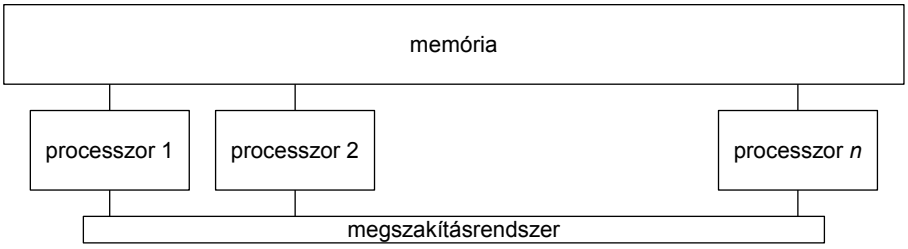
összehasonlítódik. Mindezt a hardverbe építik be, azaz operációs rendszer szempontjából ez sem többprocesszoros rendszer, így a továbbiakban ezzel az esettel sem foglalkozunk, noha bármikor amikor processzorról beszélünk, az lehet ilyen többszörös processzor is.

A fenti értelemben vett „valódi többprocesszoros rendszerek" kezelése az operációs rendszer számára nem jelent nagy, új feladatot. Mindössze az operációs rendszer megszakítási rutinjai közül a döntéshozó rutin feladatát bonyolítják kicsit, melynek már nemcsak azt kell eldöntenie, hogy melyik program fusson, hanem azt is, hogy melyik processzoron, melyik program fusson. Az operációs rendszer összes további részei változtatás nélkül is használhatóak maradhatnak.*

* Ez a gondolat csak a többprocesszoros rendszer működtetésére érvényes. A 8. fejezetben látni fogjuk, hogy ha egy-egy program futtatása során is ki akarjuk használni a rendelkezésünkre álló több processzort, akkor az operációs rendszer más funkcióit is bővíteni szükséges.

## 7.1. Egy memória – több processzor

Több processzor működhet (egy) közös memória felhasználásával. A logikailag azonos képességű processzorok mind egy-egy program utasításait hajtják végre, egymástól függetlenül, ténylegesen átlapoltan. Egy megszakítás bekövetkeztekor az ilyen gépben többféle hardver megoldás lehetséges.




*7.1. ábra. Egy memória – sok processzor.*

Az egyik lehetséges módszer szerint egy kitüntetett processzor „kapja" a megszakításokat. Ez azt (is) jelent(het)i, hogy az operációs rendszer azon az egy processzoron működik. Másik lehetséges módszer szerint a megszakítás-fogadás a processzorok között „körbejár". Ez oda vezethet, hogy egy működő processzor munkáját megszakítjuk, pedig esetleg van várakozó processzor is. A harmadik szokásos eljárás szerint a megszakítás fogadásában először a várakozó processzorok egyike jön szóba, egyébként a megszakítás-fogadás a processzorok között „körbejár". A megszakítások torlódása esetén az első módszer a megszakítási rendszer szempontjából olyan, mint az egyprocesszoros gépeknél, a második két módszerben lehetséges az is, hogy a megszakítások feldolgozása is több processzoron időben egyszerre történik. A döntési fázis – annak természeténél fogva – ekkor sem futhat egyszerre több processzoron, hiszen értelmetlen lenne több processzoron is ugyanazt a döntést meghozni (különböző döntések pedig egyenesen zavart okoznának.) A döntési rutint vagy egy kitüntetett processzoron működtetik, vagy a közös memóriahasználat jó lehetőséget ad a döntési fázis futásának jelzésére, s amíg az valahol fut, addig másutt nem indul el.

A közös memórián dolgozó többprocesszor esetében a programok semmi módon nem kötődnek a processzorokhoz, meglehet hogy egy program néhány utasítása az egyik processzoron, más utasításai másik processzorokon hajtódnak végre.

A most vázolt többprocesszoros rendszerek erősen integrált megoldása az MCM (multi chip modul) technológiával épített processzorrendszer. Többlapkás modulokból épített rendszer speciálisan nagy sávszélességű mikroösszeköttetési rendszerrel. Hasonlók ehhez MPP (massive parallel processor) rendszerek is.

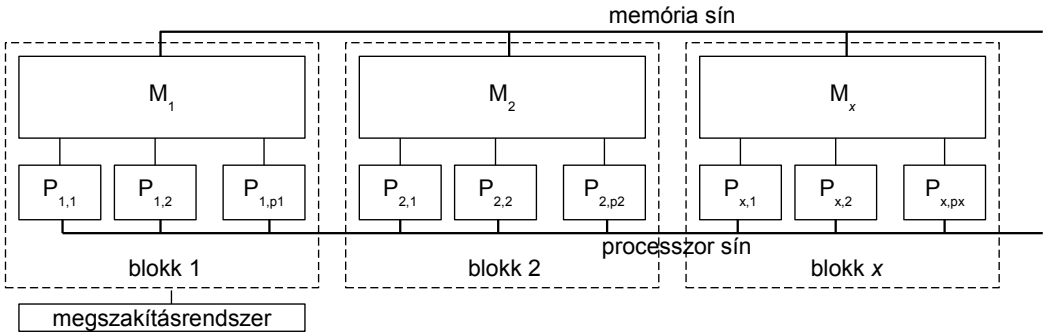
Az erősen integrált megoldás további példája a többmagos processzor. Ilyenkor egy lapkán több processzort helyeznek el. Ezek „kifelé" több, önálló processzorként látszanak. Az, hogy valóban több teljes értékű processzor kapacitását biztosítják-e, az azon múlik, hogy a lapkán mindent megtöbbszöröztek-e, vagy bizonyos funkciókat közös áramkörkészlet valósít meg.

## 7.2. Több memória – több processzor

A számítógép-építés egyik lehetséges módja, amikor többprocesszoros rendszert úgy építenek, hogy a memóriákat is többszörözik. Egy-egy memóriához egy vagy több processzor tartozik. Az egybetartozó memória-processzor(ok) egy-egy blokkot alkotnak.

Az ilyen rendszer viszonylag egyszerűen, blokkonként bővíthető. A megszakítások valamelyik (a biztosan létező első) blokkhoz futnak be. A blokkoknak nem kell egyforma kiépítésűnek lenniük. Az első blokk (az alap számítógép) a megszakításrendszert is tartalmazza. A bővítő blokkok különböző méretű memóriákat, és különböző számú processzorokat tartalmazhatnak. A programok továbbra sem kötődnek processzorhoz és blokkhoz. Az operációs rendszer (a processzorsín használatával) a processzor-állapotokat figyelve igyekszik egyenletes terheléssel dolgoztatni a teljes számítógéprendszert. Természetesen egy egyenletes terhelést követő időszakban is előállhat olyan helyzet, amikor valamelyik blokk processzora(i) szabadok, másutt pedig futásra kész programok szabad processzorra várakoznak.

> Például amikor egy adott pillanatban az operációs rendszer minden blokkban elindít programokat, de az egyik blokkba csupa gyorsan lefutó program kerül, ha most egy ideig újabb programindítási feladat nem jelentkezik, akkor megeshet, hogy a gyorsan futó programok már befejeződtek. Ekkor egyik blokk processzora(i) kihasználatlanul várakoznak, amíg másik blokkokban esetleg programok várnak processzorokra. Ilyenkor az operációs rendszer egy blokk memóriájából (a memóriasínen keresztül) egy másik blokk memóriájába töltheti át a futásra kész program pillanatnyi memória-állapotát. (A processzorsínen pedig a PSW-t, ill. a megfelelőjét tölti a megcélzott processzorba.) Ezt a program-áttöltést nem érdemes túl gyakran kezdeményezni, mert akkor az operációs rendszer jelentős idejét lekötné, a gyakoriság rendszerint paraméterezhető, általában néhány másodperces sűrűséget szoktak beállítani.




*7.2. ábra. Memória-processzor blokkok.*

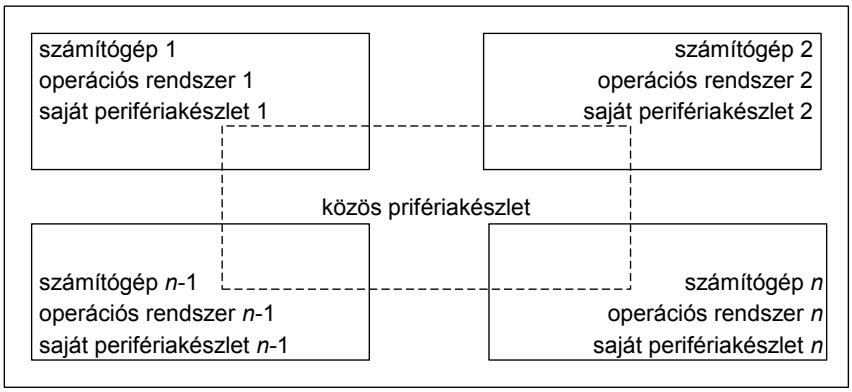
A most vázolt többprocesszoros rendszerek megvalósítása a szimmetrikus multiprocesszoros (**SMP**) processzor-architektúra. Már 2002-ben is 4-2048 processzoros SMP-rendszereket gyártottak.

Az ilyen rendszerekben nagy jelentőségű a memória(memóriák) hatékony elérése. Ennek főbb módszerei:
– **UMA** (Uniform Memory Access); egységes memória-hozzáférési idejű rendszer. Az idegen blokkbeli memória-elérése így 1,5–2 szer több időbe telik, mint a lokális memória elérése, az egységesség igénye tehát csökkenti a rendszer teljesítményét.
– A nagyobb (48–256) blokkból álló rendszerekben inkább **NUMA** (Non-Uniform Memory Access) megoldást használják. E módszer alkalmazkodik a más-más memória elérési lehetőségekhez.
– A még nagyobb (128–2048 processzoros) rendszerekben terjed a **COMA** (Cache Only Memory Access) használata. A processzorok mindig a hozzájuk tartozó (elérési lehetőségben legközelebb eső) gyorsítótárban dolgoznak. Ezzel a hardveresen támogatott replikációval érhető el a legnagyobb sebesség a memória használatában.

## 7.3. Többgépes rendszerek

A többprocesszoros rendszerek építésének további lehetséges esete, amikor több – már magában önálló számítógépből – egységes rendszert építünk. Mindegyik számítógép lehet a már korábban tárgyaltaknak megfelelően önmagában is többprocesszoros rendszer. Az így kialakított egységes rendszer számítógépei részben saját, **részben közös perifériakészlet**et működtetnek. Ezek a „leglazább" többprocesszoros rendszerek. (Több számítógépet lehet számítógép-hálózatba is kötni, ekkor azonban már nincs az összes számítógépnek közvetlenül csatlakozó közös perifériájuk. A számítógép-hálózatok tárgyalása már nem e jegyzet témája.)

A többgépes rendszer számítógépeit vezérelheti egy **operációs rendszer** (single system image). Ez a felhasználók számára egy rendszernek látszik, tényleges megoldásában a teljes operációs rendszer a gépeken működő részrendszerek együttesében dolgozik. Igen intenzív kommunikációt igényel a rendszerben működő gépek/részrendszerek között. Az ilyen együttes működés példája a fürtözéses módszer. Ezzel az eljárással tömegtermékeknek számító (tehát olcsó) számítógépekből is igen nagy teljesítményű rendszerek építhetők.




*7.3. ábra. Többgépes komplexum.*

A többgépes rendszer mindegyik számítógépét **önálló** (nem feltétlen egyforma) **operációs rendszerek** is működtethetik. Valójában az operációs rendszerek vannak egymással kapcsolatban. A felhasználó nem érzékeli, hogy többgépes rendszer szolgálja ki. Ha bármelyik géphez kapcsolt perifériáról program indítási, vagy programmal való kommunikációs feladat érkezik a többgépes rendszerhez, akkor az természetesen ahhoz az operációs rendszerhez érkezik amely a kezdeményező perifériát működteti. Ez megvizsgálja a kérést, és azt vagy teljesíti, vagy átadja valamelyik másik operációs rendszernek. Persze az operációs rendszereknek gondoskodniuk kell arról, hogy egy feladatot ne passzolgassanak körbe-körbe, hanem az valahol elinduljon. Természetesen (például speciális hardver-, szoftver-, vagy adathasználati igény esetén) rendelkezhetünk arról, hogy a program melyik gép(ek)en fusson. Ekkor a feladatelosztást nem bízzuk teljesen az operációs rendszerekre. Annak eldöntése után, hogy egy adott program hol induljon el, az a program már ahhoz az operációs rendszerhez (s ezáltal egy – persze lehet, hogy többprocesszoros – géphez) kötődik. Egy már működő programot egy másik operációs rendszer fennhatósága alá – még egyenlőtlen terhelés esetén sem – helyezhetünk át. Ebben az értelemben a többgépes rendszer a legkevésbé rugalmas a többprocesszoros rendszerek között. Tipikus példa az ilyen rendszerekre az IBM közös spool-t használó MVS operációs rendszer(családja) a JES2-vel, vagy JES3 spool-rendszerekkel.

| Processzor szám | Memória szám | Operációs rendszerek | Periféria készletek |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| $sok$ | 1 | 1 | 1 |
| $sok(*sok)$ | $sok$ | 1 | 1 |
| $sok(*sok(*sok))$ | $sok(*sok)$ | $sok$ | $részben$ 1 |
| $sok$ | $sok$ | $sok$ | $sok$ |

*7.1. táblázat. A többprocesszoros rendszerek egy csoportosítása.*

Megjegyzések:

1. Természetesen a soknak minden tényleges többszöröse is „csak” sok, itt a sok szorzataival a bővülés jellegét kívántuk érzékeltetni.
2. Bizonyos gépeken előfordul az 1 processzor-több memória szituáció is. Itt mi csak a többprocesszoros rendszerek szempontjából osztályozzuk a számítógépeket, e szempontból az 1 processzorhoz tartozó több memória nem bír különösebb jelentőséggel.

A többprocesszoros rendszereket (felületes) jellemzésükre egy táblázatba (*7.1. táblázat*) is foglalhatjuk:

A *7.1. táblázat* első sora az egyprocesszoros rendszert, az utolsó sora pedig már a számítógép-hálózatokat jellemzi. Az általunk tárgyalt, valódi többprocesszoros rendszereket a 2., 3., 4. sorok jellemzik.

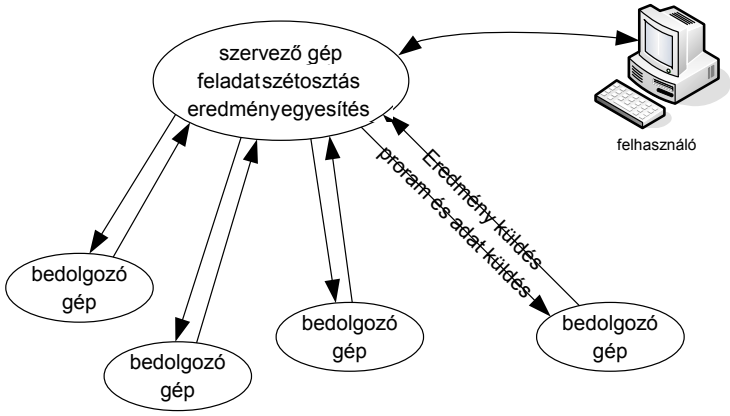
## 7.4. Számítóháló (Rács, GRID)

A számítógépek teljesítményének „egyesítésére" új elgondolás a számítóháló (grid=rács) építése. A standard, viszonylag olcsó, nagy tömegben rendelkezésre álló számítógépek (tipikusan PC-k, munkaállomások) kapacitása nagy és átlagosan csak alig kihasznált. Az ilyen számítógépeket túlnyomórészt egy felhasználó használja. Egy-egy felhasználó természetesen nem igényli a számítógépet napi 24 órában, heti 7 napon; de amikor dolgozik is a géppel, annak kapacitását sokszor csak részben használja. Elég magától értetődően adódik az ötlet, hogy ilyen számítógépek csoportjai együttes feladatokra való használata a meglévő kapacitások bizonyos értelmű összegzését jelenti, és igen nagy számítási kapacitások rendelkezésre állását biztosíthatja. A számítógépek összekötésére alkalmas – és szintén jelentősen növekedő teljesítményű – számítógép-hálózatok megadják a technikai lehetőségét a számítógép-csoportok hatékony együttműködtetésére.

A rács úgy működik, hogy mindig van egy szervező gépe (egy gép, melyen a szervező szoftver fut). A felhasználó e szervezővel áll kapcsolatban, vele közli, hogy mely programot, mely adattal kívánja működtetni.

A rács bedolgozó gépein működik a rács kliens-programja. Ha a kliens úgy érzékeli, hogy a bedolgozó gépnek van szabad kapacitása, akkor bejelentkeznek a szervezőhöz, jelzi, hogy feladatok fogadására kész. A szervező ekkor küld a bedolgozónak futtatandó programot és a program által feldolgozandó adatokat. A bedolgozó gépen, amikor a program lefutott, akkor a kliens visszaküldi az eredményeket a szervezőnek. A szervező egyesíti a bedolgozóktól kapott részeredményeket, s a teljes eredményt küldi a felhasználónak.

> Akkor is a rács bedolgozójává válhatunk, ha ez nem szándékunk, esetleg nem is tudunk róla. Előfordulnak például olyan képernyőkímélő programok, melyek éppen akkor lépnek működésbe, amikor a számítógép üzemképes, de a felhasználó éppen nem aktív. Ez a képernyőkímélő a látható tevékenységén túl egy rács kliens-programját is tartalmazhatja. Aktivizálódásakor megpróbál jelentkezni egy szervező számítógépnél, jelzi, hogy olyan gépen van, melyet a felhasználó éppen nem használ, de bekapcsolva tart, és így feladatok fogadására, programok futtatására alkalmas. Vannak ilyen pl. kulcstörő programrendszerek, melyek a jól eloszthatófeladat(ok) megoldásába annyi számítógépet vonnak be, amennyit csak elérnek.




*7.4. ábra. GRID.*

A rács olyan feladatok megoldására alkalmas igazán, amely feladat jól szétosztható. Ilyen lehet például a régi filmek felújítása, amihez kell egy alkalmas program, és a film pedig kis részekre osztható, minden bedolgozó megkapja a programot és egy-egy filmrészletet. A különböző filmrészletek feldolgozása nyilván történhet sok gépen párhuzamosan. Amikor a bedolgozó elvégezte a filmrészlet felújítását, azt visszaküldi a szervezőnek, aki a részleteket egyesíti, s a teljes felújított filmet szolgáltatja a felhasználónak. Jól szétosztható feladatokkal foglalkozunk még a *8.1. fejezet*ben is.

**Nem dedikált rács**ról beszélünk, ha a bedolgozó számítógépek elsődlegesen saját felhasználójuk igényének megfelelően dolgoznak, s csak fennmaradó kapacitásukat kínálják fel a rács közösségének. Az ilyen rács kapacitása igen nagy szélsőségek között változhat, előre biztonsággal nem tervezhető.

Meglehet, hogy egy adott nagy számításigényű és jól elosztható feladat megoldása néhány percet, máskor akár órákat, napokat is igénybe vehet, attól függően, hogy a felhasználóik mennyire veszik igénybe számítógépeiket.

Ha ez a helyzet nem elégíti ki a nagy számításigényű feladat felhasználóját (mert például a program futásideje nem lehet valamely korlátnál nagyobb, ilyen lehet a meteorológiai előrejelzés programja: ha sokat késik, eredménye már érdektelen... ), akkor kifejezetten e célra **dedikált** gépcsoportot is a **rács** rendelkezésére bocsáthatnak. Ilyenkor az egyébként olcsó, standard gépekből megbízhatóan igen nagy kapacitású rendszert építhetnek. (Természetesen a dedikált gépekhez nem feltétlenül kell monitoroknak, billentyűzeteknek, egereknek stb. is tartozniuk.)

A rácsban együttműködő gépek számának növekedésével a szervezési feladat is nő, így a felhasználók által tapasztalható teljesítménynövekedés a bevont gépek számával (pontosabban azok kapacitásával) nem egyenes arányosan nő, de sokáig növelhető, a grid kapacitása jól skálázható.

A rácsban nem dedikáltan való működtetés egy sor (ettől függetlenül is fennálló) biztonsági kérdés megbízható megoldását fokozottan igényli az operációs rendszertől. Miután a gépünkön „idegen" programok is működnek, nyilvánvaló elvárásunk, hogy az idegen program ne férkőzhessen hozzá olyan adatokhoz (fájlokhoz), memória-tartalmakhoz, a gép és rendszer olyan információihoz, melyeket nem kívánunk megosztani. általában az is igénye a nem dedikált gép felhasználójának, hogy a rácsból származó program a helyi rendszerben a legkisebb prioritást kapja, s így a helyi felhasználót semmiben ne hátráltassa.

A rácsban a szervezőprogram feladata a szabad kapacitások figyelemmel kísérése, a feladatok fogadása, elosztása, a részeredmények fogadása, egyesítése, a felhasználó számára szolgáltatása. A rács mérete (a bedolgozók száma), különösen a nem dedikált rácsban dinamikusan változhat. Akár dedikált, akár nem dedikált a rács, működése közben számos hibahelyzet fordulhat elő, a rács gépei bármikor kieshetnek (a felhasználó pl. egyszerűen kikapcsolja gépét), és a legváltozatosabb szoftverproblémák léphetnek fel. (Hardverprobléma is felléphet, de megjelenésében a rács számára szoftverproblémaként csapódik le, valamely programok nem, vagy nem megfelelően működnek.) Ilyenkor az el nem végzett feladatrészt a rács szervezője valamely más gépnek újra kiosztja. Ez nagy nyilvántartási feladatok megoldását igényli. Természetesen hálózati problémák is felmerülhetnek, de ezek áthidalása a hálózat feladata (az operációs rendszerek, s így a rács szervezőprogramja számára a hálózat egy igényre végülis kétféle választ adhat: egy adott átviteli feladatot helyesen megoldott, vagy valamilyen okból azt megoldhatatlannak minősíti.)

Találkozhatunk olyan ráccsal is, ahol a bedolgozásért valamilyen fizetséget ad a szervező, ezzel ösztönözve, hogy minél többen vállalkozzanak a bedolgozásra. Ilyenkor különösen gyakori (bár a fizetség nélküli rácsokban is előfordul), hogy a szervező gép ugyanazt a feladatot több bedolgozónak is kiosztja, és esetleg előre preparált feladatokat (melyeknek már ismeri az eredményét) is kioszt a bedolgozók között. A bedolgozóktól kapott eredményeket pedig összehasonlítja, ellenőrzi. Ezzel a bedolgozók megbízhatóságáról kíván meggyőződni, hiszen nyilvánvalóan csak annak akar „szolgáltatni" aki rendes, megbízható munkát végez.

# 8. PÁRHUZAMOSSÁG, SZINKRONIZÁLÁS

A többprocesszoros rendszerek lehetővé teszik, hogy ugyanazon a számítógéprendszeren valóban egyszerre több program fusson. Ez jó megoldás abban az esetben, ha sok felhasználói igényt sok programmal kell kielégítenünk. Nem segít azonban akkor, ha egy nagy számításigényű program futásidejét szeretnénk csökkenteni.

A futásidő csökkentésének nyilvánvaló módja, ha jobb algoritmust és/vagy jobb programozási módszert választunk, vagy gyorsabb számítógépet használunk. Ez egyik sem az operációs rendszerekre vonatkozó eljárás, ezekkel tehát itt nem foglalkozunk, jóllehet kritikus futásidejű programok gondos elkészítésével is jelentős eredményeket lehet elérni.

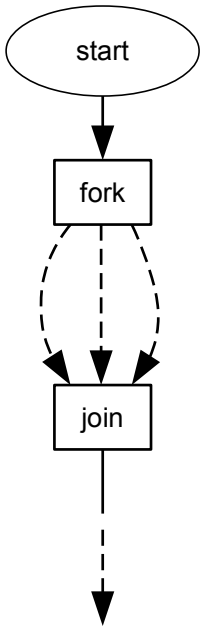
## 8.1. Programvezérelt megoldások

A továbbiakban feltételezzük, hogy a számítógéprendszerünk adott, és a program egy processzoron történő futásán programozási módszerekkel már nem tudunk javítani.

A Neumann-elvű gépeken a feladatmegoldás alapvetően szekvenciális. A program egymásután következő utasításait hajtja végre a számítógép. Részfeladatok megoldásában a tudatos emberi gondolkodás is szekvenciális, megoldunk valamit, aztán valami mást és ilyen lépések sorozatával igyekszünk eljutni a kívánt végeredményhez. Feltehetően egyebek mellett e miatt is alakult a Neumann-elvű számítógép és a programozási nyelvek szemlélete ilyen szekvenciálissá. Előfordul, hogy nagyobb feladatok egyes részfeladatai (melyek önmagukban szekvenciálisak) egymással párhuzamosan is elvégezhetők. A nagyobb feladatok megoldása olyan, amin pl. több munkás dolgozik, párhuzamosan végzett résztevékenységekből állnak össze. Ennek megfelelően a programozott algoritmusoknak is lehetnek nem szigorúan egymásra épülő, hanem párhuzamosan is végrehajtható részei.

A többprocesszoros rendszerek megjelenésével a programozási nyelvekben is megvalósították a párhuzamosíthatóságot. Ez azt jelenti, hogy olyan utasításokkal egészítették ki a nyelveket, melyek folyamatok elindítását és elkészültük bevárását teszik lehetővé. A folyamat elindítása az operációs rendszer számára nem jelent új feladatot, hiszen az megegyezik a programindítás (legősibb operációsrendszer-feladat) megoldásával. Az operációs rendszer számára új feladatot csak az jelent, hogy ha egy program jelzi (mint mindent, ezt is megszakítással), hogy most már szüksége van bizonyos, korábban indított részprogramok által szolgáltatott eredményre, akkor ezt a programot várakoztatni kell a bevárandó részprogramokból származó eredmények megérkezéséig, esetleg a részprogram befejezéséig is. Ez a technika alkalmas arra, hogy ha van szabad processzor, akkor az indított részprogram ott futhat, ténylegesen párhuzamosan a többi részprogrammal, ha nincs szabad processzor, akkor multiprogramozottan, látszólagosan párhuzamos futással a program helyesen lefuthat. Így dinamikusan kihasználhatók a párhuzamos processzorok, a programon semmit nem kell változtatni ahhoz, hogy igazodjon a rendelkezésre álló processzorok pillanatnyi számához, mindig a lehetőség szerint legjobb elosztást tudja biztosítani az operációs rendszer. Az elméleti munkákban párhuzamos programfutást indító utasítást sokszor FORK, a programok bevárását JOIN operátoroknak nevezik. Például a PL/1 programozási nyelvben gyakorlatilag is megvalósították, a párhuzamos programrészt a TASK, a várakoztatást a „WAIT FOR esemény” utasításokkal programozhatjuk.

Ha az operációs rendszert felkészítik arra, hogy programindítási kérést hálózaton keresztül másik számítógéptől (helyesebben másik operációs rendszertől) is elfogadjon, akkor így rugalmas többprocesszoros rendszer alakítható ki. Ez az operációs rendszer lényegét nem érinti, „csak" annyit jelent, hogy a hálózat felől érkező megszakítás (hasonlóan a billentyűzetről, vagy egérrel történő programindításhoz, amely szintén valójában egy megszakításfeldolgozás) hatására is adjon lehetőséget program fogadásra és indításra. Ezzel a technikával valósítanak meg például Linux-szal működő számítógépekből sokprocesszoros, párhuzamos működésű rendszereket (MPI: Message Passing Interface, PVM: Paralell Virtual Machine).




*8.1. ábra. Párhuzamosítás lehetősége a programozási nyelvekben.*

Azt, hogy egy feladat programozottan több, párhuzamosan is végrehajtható részre bontható-e, természetesen a feladat elemzésével tudjuk eldönteni. Általában hatékonyan párhuzamosíthatók például a képfeldolgozási feladatok, például egy film részekre vágható és részei párhuzamosan digitalizálhatók, javíthatók, s ezekből aztán a teljes film újra összeállítható.

Jól párhuzamosíthatók és időkorlátosságuk miatt ez nagy jelentőséget is kap, a meteorológiai feladatok (ezek tágabb értelemben a képfeldolgozás témakörbe is sorolhatók). Szintén gyakran jól párhuzamosíthatók a nagy adatbázis-műveletek, pl. egy bank összes számláján ugyanazt az algoritmust kell működtetni (kamatszámítás), akkor ez különböző számlatartományokon akár párhuzamosan is végrehajtható.

Klasszikus példák még a kulcstörő, kódfejtő eljárások, ezek is igen erősen párhuzamosíthatók (ide tartoznak az űrmegfigyelési projektek is, például a SETI (Search for ExtraTerrestrial Intelligence) projekt, mely a világűrből észlelt óriási jelsokaság elemzésével igyekszik a jelrengetegben az értelmes forrásból származókat megtalálni). Jól párhuzamosíthatók a keresések, és számos további feladat.

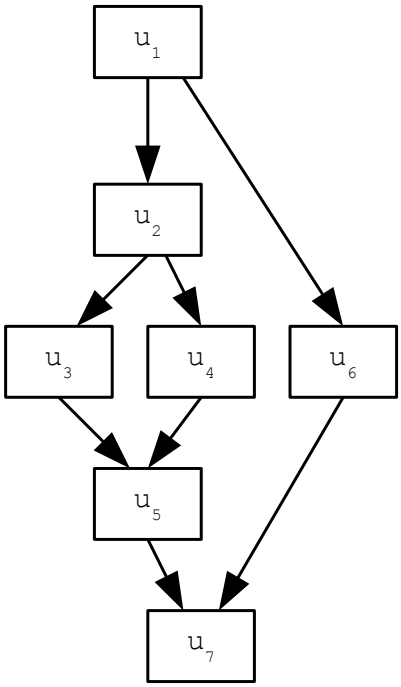
## 8.2. Automatizált megoldások

Az előző pontban tárgyalt programvezérelt megoldás jól, és a lehetőségekhez igazodva a maximális párhuzamosítást képes kihasználni akkor, ha a program megalkotója erre gondolt, és a párhuzamosítható részeket így kijelölte és megadta a bevárási pontokat is.

Ha a program készítésekor tudjuk, hogy hány-processzoros gépen fog futni, akkor annak legjobb kihasználására tudunk törekedni. Ha aztán a valóságban a program kevesebb processzoros gépen fog működni, akkor „kár volt" a sokszoros párhuzamosításba energiát fektetnünk, ha pedig a programunk olyan gépen fut majd, amely a tervezettnél több processzort működtet, akkor mégsem használja ki a hardver által immár felkínált teljes lehetőséget. Adódik tehát a párhuzamosítás – programozótól független – automatizált módja iránti igény. E kérdéskörnek

komoly elméleti vizsgálata és néhány gyakorlati eredménye messze meghaladja áttekintő tárgyalásunkat, de az alap meggondolásokat felvillanthatjuk. Vizsgáljuk meg például a következő programrészletet:

$u_1$ x:= 12;
$u_2$ y:= x/4;
$u_3$ a:= x+y;
$u_4$ b:= x-y;
$u_5$ c:= a*b;
$u_6$ d:= x+1;
$u_7$ e:= c+d;




*8.2. ábra. Precedenciagráf.*

Ha csak ennyit látunk a programból, és feltételezzük, hogy lényeges „zavaró" részei nincsenek, akkor elemezve megállapíthatjuk, hogy $u_3$ és $u_4$ csak $u_1$ és $u_2$ után hajtható végre, $u_6$ azonban már $u_1$ után, $u_2$, $u_3$, $u_4$ és $u_5$-el párhuzamosan is. A lehetséges párhuzamosításokat precedenciagráffal is szemléltethetjük. A *8.2. ábra* a példa programunk precedenciagráfja.

A gráf csúcspontjai felelnek meg az utasításoknak, az, hogy az $u_a$ csúcsból irányított él vezet az $u_b$ csúcsba, azt jelenti, hogy az $u_b$ utasítás csak az $u_a$ utasítás után hajtható végre.

A párhuzamosíthatósági feltételek jól megfogalmazhatók.

**Bernstein-tétel: Egy program két, $u_a$ és $u_b$ utasítása csak akkor hajtható végre konkurens módon, ha:**

$$\mathbf{R(u_a) \cap W(u_b) = \emptyset,}$$
$$\mathbf{W(u_a) \cap R(u_b) = \emptyset,}$$
$$\mathbf{W\,(u_a) \cap W\,(u_b) = \emptyset.}$$

Ahol $\mathbf{R(u_a)}$ jelenti mindazon változók halmazát, melyeket az $u_a$ utasítás felhasználhat (olvasási, „read" halmaz), $\mathbf{W(u_a)}$ jelenti mindazon változók halmazát, melyek értékét az $u_a$ utasítás megváltoztathatja (írási, „write" halmaz).

Az ilyen tulajdonság a programokon belül viszonylag jól algoritmizálhatóan megkereshető és kihasználható. Azaz a program akkor is párhuzamosítható, ha írója erről eredetileg nem gondoskodott.

Látható azért az is, hogy a feladat a példában szereplőnél bonyolultabb, mert nem lehet a változóneveket pusztán karakteres azonosítójuk alapján biztosan megkülönböztetni, ugyanis a legtöbb programozási nyelv lehetőséget ad ugyanazon memóriaterületre (ahol a változót elhelyezzük) többszörös és átlapolt módon hivatkozni. Így forrásprogramokban azt külön algoritmu-

soknak kell eldönteniük, hogy két látszólag különböző változó között van-e valamilyen „rejtett" kapcsolat. Az operációs rendszer szempontjából – mely gépi kódú programokkal foglalkozik – a helyzet azért nem reménytelen. A gépi kódú programban már semmi jelentősége az eredeti forrásprogramban használt szimbólumoknak. A gépi utasítások címrészének elemzésével eldönthető, hogy azonos memóriaterületre hivatkoznak-e vagy sem. Az automatizálás jó lehetősége, hogy pontosan akkora párhuzamosságot keres, amekkorát a hardver éppen lehetővé tesz. Változó körülmények között is jól szolgálja a hardver legjobb kihasználását, minden processzort felhasználhat és „nem is lő túl a célon".

## 8.3. Kliens-szerver megoldások

Összetett feladat részeinek párhuzamosítását akkor sem mindig automatizálhatjuk, ha erre rendszerünk valamilyen segítséget ad. Vannak olyan feladatok, melyeket ésszerű, hasznos párhuzamosítani, de párhuzamos részei célszerűen csak kitüntetett helyeken (gépeken) futtathatók.

Egy példával megvilágítva: legyen a feladat egy nagy könyvtárban történő keresés-válogatás és az eredmények valamilyen grafikus feldolgozása.

A könyvtár könyvállományát tartalmazó nagy adatbázist célszerűen egy példányban tartják egy szerveren. Ha most valahol egy másik gépen feltesszük a válogatásra vonatkozó kérdést, akkor lehetséges, hogy a könyvtári adatbázis minden egyes tételét hálózaton keresztül eljuttatjuk a válogató géphez, és az majd kiválogatja azokat a tételeket, melyekre éppen szüksége van. Ezt követően elvégzi velük a további feldolgozást és elkészíti a grafikus megjelenítést. Ez a megoldás nem szerencsés, mert ha igen nagy (akár több millió) könyv közül a válogatásnak csak néhány száz felel meg, akkor a sokmillió adatot eljuttatjuk a válogatást végző géphez, és igen nagy részét csak azért, hogy eldőljön róla, hogy érdektelen. Ez nyilván nagy hálózati forgalmat generál, költséges és lassú.

Másik lehetséges megoldás, hogy a feladatot ott oldjuk meg, ahol az adatbázis van. Mivel azonban továbbfeldolgozást, grafikus ábrázolást is meg kell oldani, és lehet, hogy sok más helyen velünk egy időben szintén hasonló feladatot fogalmaztak meg, így az adatbázist kezelő kiszolgáló gépet terhelné sok (például az internetet használva akár sok millió, lényegében egy időben megjelenő igény kiszolgálása, amint az az utóbbi olimpiák alatt megtörtént) munkaigényes grafikus feldolgozás is. Ehhez vagy nagyon nagy kapacitású (következésképpen drága) rendszer kell, vagy csak lassan képes kiszolgálni az igényeket, ez sem szerencsés megoldás tehát.

A célszerűnek tartott megoldás ilyen jellegű feladatokra az, hogy a feladatot oly módon osszuk részekre, hogy minden részfeladat (ha lehet párhuzamosan is) ott legyen elvégezhető, ahol az a legcélszerűbb. A példánkban szereplő feladat megoldásához legalább két (rész) programot kell elkészíteni. A felhasználói gépen futó részt oly módon írjuk meg, hogy az csak a válogatási szempontokat küldje el az adatbázist tartalmazó géphez, az ottani programrész válogassa ki a most éppen érdekes adatokat, majd ezt a „nyersanyagot" küldje vissza a felhasználói géphez, aki a további feldolgozást, megjelenítést már maga végezze el. Nos az ilyen feladatmegosztást nevezik **kliens-szerver** megosztásnak. Ekkor felhasználói szempontból *egy* feladatot oldanak meg, de gépek között megosztva, ha lehet párhuzamosítva is, úgy, hogy a gépek azon feladatrészeket végzik el, amelyeket azon a gépen célszerű. Ezzel idő, pénz takarítható meg és más szempontokból is al-

kalmas megoldás (adathozzáférés-ellenőrzés, kézbentarthatóság stb.) A kliens-szerver megoldások részben operációs rendszerrel összefüggő témák, de erősen átnyúlnak mind a programozási, mind a hálózati témakörökbe is, ott is találkozunk speciális kérdéseikkel.

Az adatbázisok alkalmazása szinte mindig kliens-szerver jellegű. Akár beágyazottan, akár általános SQL-felületet nyújtó programmal használjuk az adatbázist a felhasználói programunk tipikusan egy másik számítógépen működő adatbázis szerver szolgáltatásait veszi igénybe. A felhasználói alkalmazás a kliens, amely SQL-utasításokat küld a szervernek, az végrehajtja az adatbázisban a kért akciót, majd annak eredményét visszaküldi a kliens alkalmazásnak, amely a kapott eredményt tovább feldolgozhatja. A megvalósítás egységes, még akkor is ezt alkalmazzák, ha kivételesen a szerver és a kliens futtató gépe ugyanaz a számítógép.

# *9. HOLTPONTMENTES VEZÉRLÉS*

A program egy előre megadott algoritmus. Ugyanazon program nagyon sokféleképpen futhat, az aktuális adatai által meghatározott módon. A számítógépen programok futnak, egy-egy konkrét – algoritmusuk és aktuális adataik által meghatározott – módon. Egy program egy konkrét futását **folyamat**nak nevezzük. A számítógépünkben tehát folyamatok működnek. Összességében a működést egy folyamatokból álló rendszernek is tekinthetjük. Az operációs rendszer feladata, hogy a folyamatokból (programok konkrét futásából) álló rendszer működéséről gondoskodjon. A folyamatok működésük közben erőforrásokat igényelhetnek. Ez konfliktushelyzeteket teremthet. Amíg a folyamatok más-más erőforrásokat használnak, vagy (részben) ugyanazon erőforrásokat, de osztott módon használják, addig nincs is konfliktus. Előfordul azonban, hogy az erőforrás kizárólagos használata szükséges.

Gondoljunk például arra, ha egy számlákkal foglalkozó rendszerben ugyanazon számla adatait megengednénk időben átlapoltan két helyről módosítani. Tegyük fel például, hogy a számlán induláskor 10 (Ft) van. A pénzintézethez erre vonatkozóan két befizetés érkezik, egy 100 és egy 1000 (Ft). Előfordulhat, hogy e két számlát két dolgozó egyszerre veszi kézbe és igyekszik könyvelni. A könyvelő program előveszi a jelenlegi számlaadatot, azaz a 10-et, (egyik folyamat is, másik folyamat is). Ezután amelyik folyamat picit gyorsabban fut (mert pl. a dolgozó gyorsabb) az elvégezvén feladatát visszaírja a mondjuk 1000-el megnövelt, vagyis 1010 adatot. Röviddel később a másik folyamat is eljut a visszaírásig, azaz az általa (is) beolvasott 10 et megnöveli 100-al, így visszaír 110-et. Így aztán a számlán a helyes 1110 helyett 110 marad eredményül. Mindkét folyamat önmagában hibátlanul, és így hibaüzenet nélkül futott, az eredmény mégis rossz. Ennek oka az együttes futásuk volt.

Konfliktus akkor keletkezhet, ha már más(ok) által használt erőforrást egy folyamat kizárólagosan kíván használni.

A folyamatok és erőforrások viszonya a következő lehet:
– a folyamat és erőforrás elkülönültek,
– a folyamat bejelentette igényét erőforrás használatára (osztott vagy kizárólagos módon),
– a folyamat az erőforrást használja,
– a folyamat bejelenti az erőforrás felszabadítását.

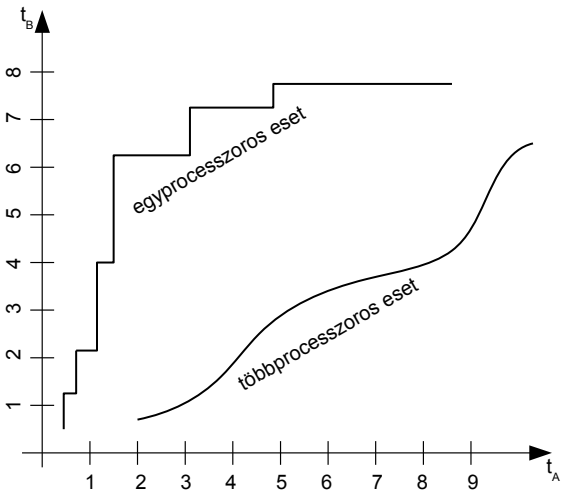
Az erőforrások kiosztása a folyamatok között az operációs rendszer feladata. Így a fentebb emlegetett bejelentések az operációs rendszernek szólnak. Ha egy folyamat egy adott erőforrást már használ (megkapott az operációs rendszertől), akkor ugyanazt az erőforrást másik folyamat csak akkor kaphatja meg az előző folyamattal egyszerre, ha mindketten osztott használatot kértek.

**Megvalósítható állapot**nak a folyamatokból álló rendszer azon állapotát nevezzük, amelyben a folyamatok és erőforrások viszonya ellentmondásmentes. Más szavakkal az olyan állapot a megvalósítható, melyben minden folyamat megkapta az általa éppen igényelt erőforrásokat az általa igényelt módon, vagy várakozik másik folyamat által használt erőforrásra. Ennek megfelelően az olyan állapotot, melyben a folyamatok és erőforrások egymáshoz rendelése ellentmondásos, **megvalósíthatatlan állapot**nak nevezzük. Megvalósíthatatlan állapot például az olyan ál-

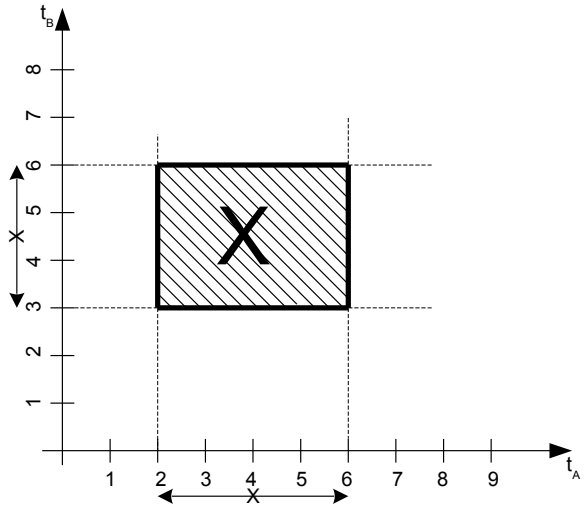
lapot melyben két folyamat ugyanazt az erőforrást kapná meg egyszerre kizárólagos használatra. Az operációs rendszer végzi az erőforrás-kiosztást, s így a folyamatokból álló rendszer (az éppen működő programok együttese) nem kerülhet megvalósíthatatlan állapotba. Ezzel tehát nincs gond, probléma úgy jelentkezhet, hogy a folyamatokból álló rendszer működése során olyan helyzethez is eljuthat, melyben a folyamatok erőforrásokra várakozásában kör alakul ki. Ekkor ezek a folyamatok végtelen ideig kellene, hogy várakozzanak. Ebből a helyzetből kár nélkül nem tudunk kikeveredni. Az ilyen helyzetet nevezzük **holtpont**nak.

A továbbiakhoz szükséges még egy fogalom meghatározása. A folyamatok működésükkor (azaz a programok futásukkor) nem folytonosan futnak, hanem sok-sok kisebb-nagyobb időszeletnyit működnek, majd – megszakítások hatására – várakozni kényszerülnek. Beszélhetünk azonban egy „tiszta" futásidőről, arról az időről, amely a program utasításainak végrehajtási ideje. Ezt a folyamat **sajátidej**ének nevezzük. Szokás ezt **CPU-idő**nek is nevezni. A program elindulása és befejeződése közötti teljes (felhasználó által érzékelt) naptári időt pedig **start-stop**, vagy **futási idő**nek is szokták nevezni. Egy adott folyamat (program konkrét adatokkal) ugyanazon a számítógéprendszeren mindig lényegében ugyanannyi *saját idő* alatt fut le, de a futási körülmények változása miatt a *futási ideje* szinte biztos, hogy mindig más és más.

Egyetlen folyamat – ha a programot jól írtuk meg – önmagában nem juthat holtpontra. A holtpont kialakításában legalább két folyamat szerepel. Az *A* és *B* két folyamatból álló rendszer állapotainak és a holtponttal kapcsolatos helyzetek szemléltetéséhez olyan koordinátarendszert használunk, melynek egyik tengelyén az *A*, a másik tengelyén a *B* folyamat sajátidejét mérjük. Ebben a koordináta-rendszerben a $(t_A, t_B)$ pont a folyamatokból álló rendszer egy állapotát jelenti, azt, hogy az *A* folyamat éppen sajátidejének $t_A$ , a *B* folyamat sajátidejének $t_B$ pillanatához jutott el. Az állapotot jelképező pont az idő múltával elmozdulhat, aszerint, hogy a folyamatok sajátidejükben éppen hol tartanak. Az állapotot jelképező pont mozgása által kirajzolt görbe csak monoton növekvő lehet (hiszen az idő csak múlhat). A *9.1. ábrán* láthatóhoz hasonlóan, egyprocesszoros rendszerben ez a görbe csak vízszintes és függőleges szakaszokból állhat (aszerint, hogy éppen melyik program fut a processzoron annak a sajátideje halad, növekedik, a másik folyamat áll, tehát sajátideje változatlan), többprocesszoros rendszerben bármilyen monoton növekvő görbe lehet, hiszen mindkét program sajátideje egyszerre is „múlhat".




*9.1. ábra. Folyamatokból álló rendszer állapotgörbéje.*




*9.2. ábra. A megvalósíthatatlan állapotokat jelentő téglalap.*

Ebben a koordináta-rendszerben valamely erőforrás kizárólagos használatának igénye miatti megvalósíthatatlan állapotokat téglalapok jelképezik. A *9.2. ábrán* olyan helyzetet látunk, amikor az *A* folyamat valamely *X* erőforrást sajátidejének 2–6 időpillanatai között kíván kizárólagosan használni, és ugyanezt az erőforrást a *B* folyamat az ő sajátidejének 3–6 időpillanatai között szintén kizárólagosan kívánja használni. Ekkor az ábrán satírozott téglalapba nyilvánvalóan nem juthat a folyamatokból álló rendszerünk állapotát jelképező görbe, az ugyanis azt jelentené, hogy olyan helyzet lenne, amelyben mind az *A*, mind a *B* folyamat egyszerre kizárólagosan használná az *X* erőforrást. Arról könnyen gondoskodik az operációs rendszer, hogy a megvalósíthatatlan állapotokat elkerülje, hiszen ha valamely folyamat valamely erőforrást igényel (egy megszakítás segítségével az operációs rendszertől), akkor a megfelelő táblázataiban ellenőrzi, hogy az igényelt erőforrás szabad-e? Ha nem, akkor az ezt kérő folyamatot várakoztatja. Az erőforrások felszabadításakor (ezt szintén megszakítással hozzák a folyamatok az operációs rendszer tudtára) a rendszer azt is megvizsgálja, hogy a most felszabadított erőforrásra várakozott-e valamely más folyamat. Ha igen, akkor az erőforrást ahhoz rendeli, és futtathatónak minősíti. Ez az ábrán azt jelenti, hogy a folyamatokból álló rendszerünk állapotát jelképező görbe a megvalósíthatatlansági területekre nem lép be, annak legfeljebb a határáig juthat el, s a határon mozogva a megvalósíthatatlansági területet végül elkerüli.

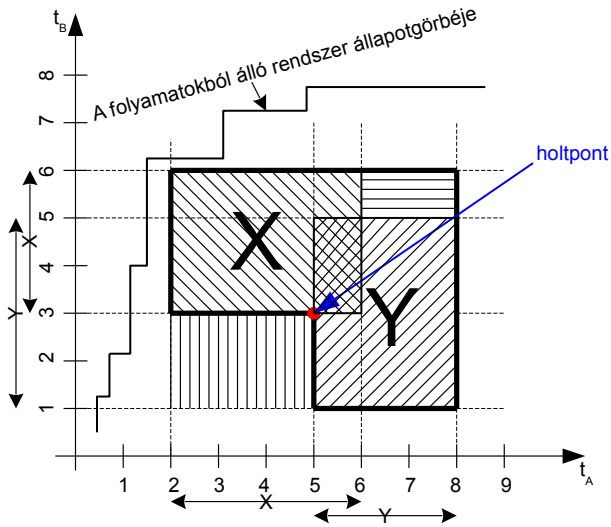
## 9.1. A holtpont

Visszatérve fő gondolatmenetünkhöz, vizsgáljuk meg a holtpont kialakulását és igyekezzünk következtetni arra, hogy hogyan tudjuk e problémát megoldani vagy elkerülni.

Egy példával illusztrálva a holtpontot: tegyük fel, hogy csak két folyamatból álló rendszert vizsgálunk. Továbbá tegyük fel, hogy az *A* folyamat az *X* erőforrást saját idejének 2. és 6. időpillanata között, az *Y* erőforrást saját idejének 5. és 8. időpillanata között kívánja kizárólagosan használni, a *B* folyamat pedig az *X* erőforrást a saját idejének 3–6. időszakában, az *Y* erőforrást saját idejének 1–5. időszakában kívánja kizárólagosan használni.

| | „$A$” folyamat | „$B$” folyamat |
|---|---|---|
| „$X$” erőforrás | 2-6 | 3-6 |
| „$Y$” erőforrás | 5-8 | 1-5 |

Ekkor, ha az *A* folyamat eljutott sajátidejének 5., *B* folyamat pedig sajátidejének 3. időpillanatához, akkor e két folyamatunkból álló rendszer holtpontra jutott. Ugyanis az *A* folyamat nem tud tovább működni, mert kizárólagosan szeretné használni az *Y* erőforrást, melyet ekkor a *B* folyamat már használ (hiszen sajátidejének már 3. pillanatában járunk, és a 1. idő óta az *Y* erőforrást már kizárólagosan megkapta, de még nem engedte el, arra ugyanis csak a sajátidejének 5. pillanatában fog sort keríteni). Ezért az *A* folyamat várakozásra kényszerül, ki kell várnia amíg az *Y* erőforrást most használó *B* folyamat azt felszabadítja. De ugyanekkor a *B* folyamat is várakozásra kényszerül, mert ő meg az *X* erőforrást szeretné megkapni, melyet ekkor – a már várakozó – *A* folyamat használ kizárólagosan. Mindkét folyamat várakozik, arra, hogy a másik haladjon tovább és szabadítsa fel az általa használt és most a másik számára is szükséges erőforrást, a várakozásuk miatt az erőforrás felszabadítására nem kerül sor, tehát a várakozás örökké tarthatna, a folyamatok soha nem fejeződhetnének be, ezt nevezzük *holtpont*nak. A holtponti helyzetből csak valamely folyamat erőszakos befejezésével („kilövésével”) lehet kikeveredni és ez mindenképpen veszteség. A nem normálisan befejeződött program használta a számítógéprendszert, esetleg adatmódosításokat hajtott végre, emiatt ismétlése valamilyen visszaállítást, előkészítést igényelhet, s ezzel mindenképpen kár keletkezett.




*9.3. ábra. A holtpont kialakulása.*

A *9.3. ábra* a rendszer állapotait mutatja a már említett példa adataival. Az ábrán a ferdén satírozott területek az *X* (a (2,3) és (6,5) pontok által kijelölt téglalap) illetve az *Y* (az (5,1) és (8,5) pontok által kijelölt téglalap) erőforrásigények miatti megvalósíthatatlan állapotokat jelzik. A vízszintesen satírozott terület (a (6,5) és (8,6) pontok által meghatározott téglalap) elvileg ugyan már nem lenne megvalósíthatatlan állapotok területe, gyakorlatilag azonban, minthogy az állapotgörbe monoton nő, ide sem juthat bele. Végülis a vastagon keretezett rész a teljes meg-

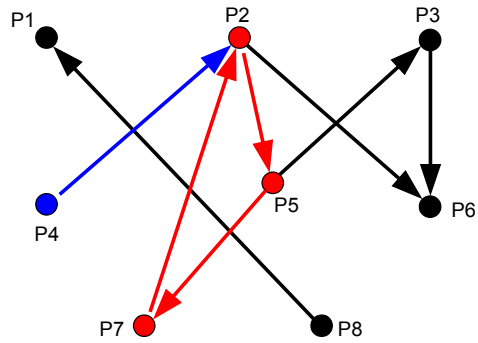
valósíthatatlan állapotok területe. Az igazi gond az ábrán függőlegesen satírozott (a (2,1) és (5,3) pontok által kijelölt téglalap), úgynevezett **holtponti terület**tel van. E terület jobb felső sarka (példánkban az (5,3) pont) a holtpont. A holtponti terület veszélye az, hogy ebben a területben még zavartalanul működhetnek a folyamatok, de (merthogy jobbról és felülről már megvalósíthatatlan területek határolják) már biztosan, „sorsszerűen" a holtponthoz fog jutni a (folyamatokból álló) rendszer. Első gondolatra elvárhatnánk az operációs rendszertől, hogy gondoskodjon a holtponti helyzetek elkerüléséről. Ez a feladat a(z eddig egyszerűnek látott) döntéshozó rutin dolga lehetne, neki kellene úgy kiválasztania a futó a programokat, hogy elkerülje a holtponti helyzeteket. Ehhez azonban a holtponti területet is el kellene kerülnie. Ez az ábrán egyszerű, valóságban azonban ehhez az operációs rendszernek előre kellene látnia, hogy mely folyamat mely erőforrást fogja igényelni, és mikor fogja felszabadítani. A futó programok viselkedését azonban a bemenő információik befolyásolják. Azt például nem lehet előre megjósolni, hogy egy majdani párbeszéd során a felhasználó milyen adatot ad, vagy milyen döntést hoz. Így az operációs rendszertől sajnos nem várható el a holtpont megbízható elkerülése.

## 9.2. A holtpont kialakulásának felismerése

Amint láttuk, a holtpont elkerülése (jövőbelátás híján) nem várható el az operációs rendszertől. A már kialakult holtpont viszont egy már előállt állapot, ennek felismerését, jelzését joggal elvárhatjuk az operációs rendszertől.

Az eddigi példában két program (folyamat) és két erőforrás viszonyán vizsgáltuk a holtpont kialakulásának lehetőségét. Ez a legkisebb szereplőszámú holtpont. Holtpont természetesen bonyolultabb módon is kialakulhat, több program, több erőforrásra való várakozásakor igen változatos helyzetekben.

A holtpont nem jelenti a rendszer megállását („lefagyását", például a „kék halál"-t). Attól, hogy kettő vagy több program olyan várakozási helyzetbe került, hogy olyan programok továbbműködésére várnak, melyek szintén várakoznak, s ebben a várakozási láncban körök alakultak ki, tehát végtelen várakozási helyzet (=holtpont) alakult ki, nos ettől más programok még zavartalanul működhetnek. „Ránézésre" tehát a számítógépünk normális működő képet mutat, vannak futó programok és vannak várakozó programok, mint egy átlagos, „normális" helyzetben. Az tehát külön vizsgálat tárgya, hogy az egymásra várakozások rendszerében kialakult-e holtpont. Ha az elindított programokat pontokkal, a várakozási helyzeteket nyilakkal szemléltetjük (a nyíl kezdőpontja a várakozó program, végpontja az a program, amely által lefoglalt erőforrásra, azaz a program futására vár, mely során a program a kérdéses erőforrást majd felszabadítja), akkor egy irányított gráfot kapunk. Programok akkor jutnak holtpontra, ha ebben a gráfban köröket találunk.




*9.4. ábra. A holtpont felismerése.*

Hogy a rendszerben kialakult-e holtpont, azt a várakozási gráfban körök keresésével tudjuk eldönteni. A *9.4. ábrán* látható példában egyetlen kört: P2–P5–P7 találunk. Ez holtpontot jelent. Megfigyelhetjük azt is, hogy emiatt nemcsak a körben résztvevő három program várakozik végtelen ideig, hanem a P4 is. P4 bár a körben nem szerepel, de valamelyik (példánkban a P2) körben szereplő, már holtpontban lévő programra vár.

A holtpont kialakulását az operációs rendszerek általában a várakozási gráfban a körök keresésével figyelik. A gráfban új nyíl akkor keletkezik, ha valamely folyamat (futó program) új kizárólagos igényt jelent be egy erőforrásra, s azt már más folyamat birtokolja, vagy nem kizárólagos igényt jelent be, s az erőforrást más folyamat már kizárólagosan birtokolja. A várakozási gráfban kör akkor keletkezhet, ha új nyíl jelenik meg. Az operációs rendszer tehát vagy minden új erőforrásigény bejelentésekor ellenőrzi, hogy kialakult-e holtpont, vagy bizonyos – az átlagos erőforrásigénylési gyakoriságtól nagyobb – időközönként végzi el* a vizsgálatot.

## 9.3. A holtponti helyzet megoldása

Ha holtpont kialakulását érzékelte az operációs rendszer (vagy a felhasználó), akkor a várakozási körben résztvevő (nem az összes végtelen várakozásra sodródott!) folyamatok valamelyikét automatikusan megszünteti, vagy a rendszer működtetőjét (operátort/felhasználót) tájékoztatja a helyzetről, s kéri valamely a várakozási körben résztvevő folyamat megszüntetését. Általában ez utóbbi eljárás a kisebb veszteségű, bár emberi közreműködést igényel. A folyamatok megszüntetése, programok „kilövése” veszteség, hiszen kárba vész az általa eddig használt processzoridő, memória stb., a programot meg kell ismételni, ami további előkészítési, ** visszaállítási tevékenységet igényelhet (ha egy program adatokat módosít, s félbe kell szakítani, akkor ismétlése előtt a futását megelőző állapotot kell visszaállítani). Általában a legkisebb veszteséget olyan folyamat megszüntetése jelenti, amely ismétlése a legkisebb előkészítést igényli, s eddig a leg-

* Igen sajnálatos, hogy ezt a nem különösebben bonyolult, de igen fontos vizsgálatot néhány igen népszerű operációs rendszerbe sem építették be, s így semmi jelzést nem kap a felhasználó a holtpont kialakulásáról.

** Vannak operációs rendszerek, például az IBM MVS családja, melyben a programok futtatását ellenőrzőpont képzéssel is kérhetjük. Ezzel jelentősen csökkenthető az ismétlési veszteség, ugyanis visszaléphetünk olyan ellenőrzőpontig, amely megelőzi a holtpontot okozó erőforrás lefoglalását, az tehát felszabadítható, a holtpont megszüntethető.

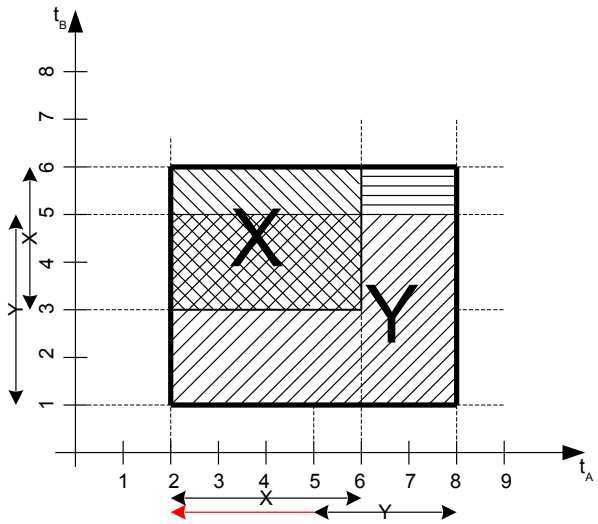
kevesebb ideig működött. Ennek kiválasztására az operációs rendszer is rendelkezhet algoritmussal, de a kezelőre/felhasználóra is bízhatja, esetleg a javaslatának feltüntetésével. A holtponti helyzetből való kilábalás automatizálásának egyik módja az éheztetésnek nevezett döntési módszer. Ez azt jelenti, hogy nem engedjük a folyamatokat sokáig várakozni. Minden folyamathoz a felhasználó (az operációs rendszer) egy maximális megengedett várakozási időt rendel. Ennek elérésekor a folyamatot az operációs rendszer várakozási idő túllépése miatt „erőszakkal" befejezi. Ez természetesen szintén veszteség, az operációs rendszer dolgát viszont egyszerűsíti, mert nem kell figyelni a holtpont kialakulását. A módszer hátránya viszont, hogy olyan programok kilövéséhez is vezethet, melyek nincsenek holtponton csak valami más okból hosszú várakozásra kényszerültek. E módszer sem nevezhető tehát minden szempontból jó megoldásnak. Ha a holtponti helyzet kialakult, akkor már mindenképpen csak veszteség árán tudunk abból kikeveredni. Igen fontos tehát a holtpont elkerülése.

## 9.4. A holtponti helyzet elkerülése

Az előző vizsgálatainkban megállapítottuk, hogy ha a holtponti helyzet kialakult, akkor már mindenképpen csak veszteség árán tudunk abból kikeveredni. Igen fontos tehát a holtpont elkerülése.

Ha átgondoljuk, hogy hogyan is alakul ki holtpont, felismerhetjük, hogy ez az olyan helyzetekben következhet be, amikor egy folyamat már birtokol erőforrást, és újabb megszerzésére vár. Ekkor kialakulhat várakozási kör a folyamatok között. Látható tehát, hogy az olyan helyzeteket kell elkerülni, hogy egy erőforrás megszerzését követően próbáljon a folyamat újabbat is megszerezni. A ilyen időszakokat a *9.6. és 9.7. ábrán* „kritikus"-ként jelöltük. Ha ezen időszak alatt a vizsgált programmal együtt futó más program lefoglalja azt (a példákban a C jelű erőforrást), melyet a vizsgált program is szeretne majd lefoglalni, akkor holtpont alakulhat ki.

A holtponti helyzetek egyszerű és biztos elkerülése megoldható alkalmas programozási módszerrel. Nevezetesen, ha egy program akkor, amikor egy erőforrásra kizárólagos használattal szüksége lesz lefoglalja az éppen igényelt erőforrással *időben átlapoltan használni kívánt összes* erőforrás, továbbá a korábban kizárólagos használatra lefoglalt erőforrásokat nyomban felszabadítja ha a program ezen fázisában már nincs tovább szüksége rá. Ez az átalakítás a *9.1. fejezet*ben bemutatott esetben azt jelenti, hogy például az *A* folyamat az *Y* erőforrást már a sajátidejének 2. időpillanatában igényelje (az ekkor igényelt *X*-el együtt). Ezáltal az *Y* erőforrás miatti megvalósíthatatlan állapotok területét jelképező téglalap megnő ugyan, de már nem lesz sehol olyan pont és terület sem, melyet jobbról és felülről megvalósíthatatlanági területek határolnának, azaz eltűnt a holtpont, ez volt a célunk.

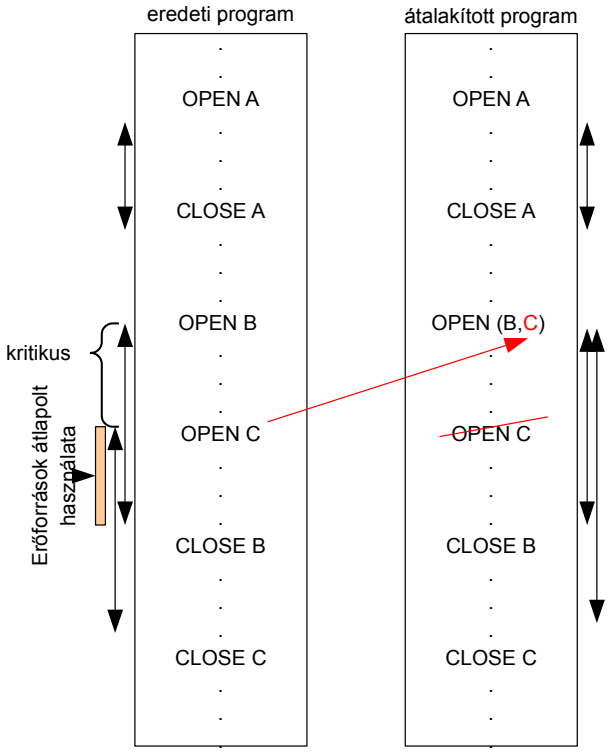



*9.5. ábra. Megfelelő erőforrás-foglalással a holtpont megszűnik.*

Így a program a ténylegesen szükséges időnél tovább tart erőforrásokat lekötve, erre lehet, hogy várakoznia kell, s ő másoknak okoz majd várakozást, de biztosan nem jut holtpontra. A holtpont kialakulásának – elkerülésének – megelőzésének tanulmányozásából tehát programozási következtetést vontunk le. Szerencsére ez a programjaink algoritmusát lényegesen nem befolyásolja, a programozásban lényeges többletterhet nem okoz, viszont az igen súlyos holtponti problémát megoldja.

Egy példán megvizsgálva, hogy milyen átalakítást kell tennünk a programunkon, hogy az ne sodródhasson holtpontra:

Tegyük fel, hogy a program megírásához használt programozási nyelvben az erőforrás-igénylő utasítás az OPEN, az erőforrás-felszabadító utasítás a CLOSE. A programot – még holtponti kérdésekkel nem foglalkozva –„tisztán” a megvalósítandó feladatra koncentrálva elkészítjük, ennek sémáját a *9.6. ábra* bal oldalán ábrázoljuk:



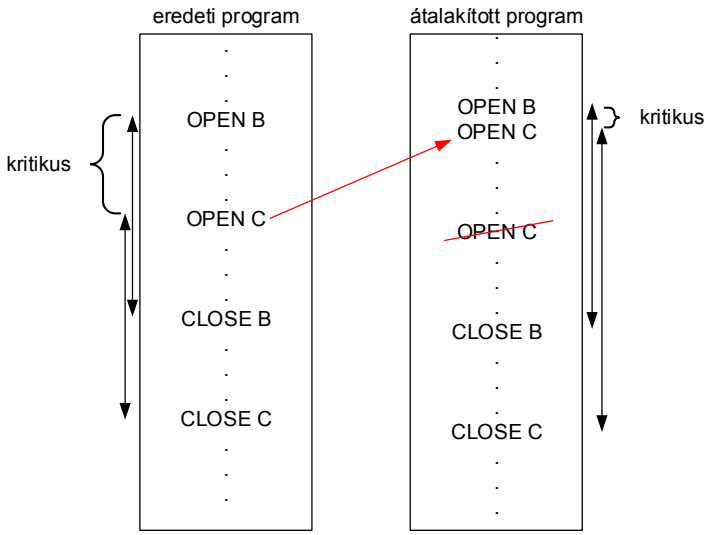

*9.6. ábra. A program átalakítása holtpontveszély-mentessé.*

Azt tapasztaljuk, hogy a programban van olyan fázis, amikor a B és C erőforrásokat párhuzamosan használni kívánja. Ekkor a holtpont elkerülése céljából az OPEN C utasítást az OPEN B utasítással egyesítenünk kell.

A *9.6. ábra* jobb oldala a holtpont elkerülésére átalakított programsémát mutatja. Látható, hogy az átalakítás a program tényleges algoritmusán semmit nem változtat, s az átalakítás könnyen végrehajtható.

Meg kell jegyezni, hogy ha az alkalmazott programozási nyelv (és az alkalmazott operációs rendszer együttműködése), azaz a programfejlesztési és futtatási környezet nem teszi lehetővé, hogy egy atomian (garantáltan együtt végrehajtott utasításcsoporttal) működő utasítással több erőforrást is lefoglaljunk, akkor az a környezet nem alkalmas olyan programok megírására, melyek garantáltan nem jutnak holtpontra. A *9.7. ábrán* látható átalakítás – ha az adott programozási környezet nem teszi lehetővé a két utasítás együttesen atomi végrehajtását akkor – nem garantálja a holtpont elkerülését, mert a két OPEN utasítás végrehajtása között a program futása megszakadhat, a B erőforrást már megszerezte, a C-t majd később próbálja megszerezni, s ez éppen holtponthoz vezethet.

Igaz, ezzel az átalakítással a *9.5. ábra* holtponti területe, *9.7. ábra* kritikus időszaka lényegesen kisebb lett, tehát a holtpontra sodródás esélye jelentősen csökkent, de nem szűnt meg.




*9.7. ábra. A holtpont-veszély csökkentése alkalmatlan nyelvek eetén.*

Ha módunkban áll programozási környezetet választani, ilyen fejlesztési környezetet ne használjunk!

*A holtpontmentességet a program tetszőleges számú hibátlan tesztelő, ellenőrző futtatása nem bizonyítja!* Egyébként hibátlan programok esetében a holtpontot több program együtt tudja kialakítani. Bármennyi hibátlan futás csak azt jelenti, hogy mindig olyan szerencsés működési környezetben futott, ahol éppen nem alakult ki holtpont. A számítógépek teljesítményének növekedése, és a hálózatok felhasználói számának növekedése oda vezet, hogy nagyon sok programot együttfuttató rendszerekben működnek programjaink, itt a holtpont kialakulásának esélye nő, helyes tehát, ha programjaink elkészítése során e veszélyt figyelembe vesszük, és igyekszünk elkerülni.

## 9.5. Erőforrások feltételes megszerzése

Eddig olyan programozási módszerekkel foglalkoztunk, amikor az erőforráskérés feltétlen volt. Ez azt jelenti, hogy a program kéri az erőforrást az operációs rendszertől, s addig vár, amíg meg nem kapja, vagy az operációs rendszer kilövi, mert a kiéheztetés során elérte a maximálisan megengedett várakozási időt. (Ha megfelelő fejlesztési környezetben megfelelően programoztunk, akkor holtpont már nem alakulhatott ki.) Ez a módszer kötegelt programok esetében általában megfelelő lehet, az ilyenek futásideje általában nem kritikus, nem okoz gondot, ha esetenként a program futása közben hosszabb várakozásra kényszerül. A párbeszédes programok esetében azonban hosszú várakozások nem megengedettek. A felhasználót – joggal – zavarhatja ha a program hosszasan várakozik, különösen, hogy – mivel nem fut – a várakozás okáról sem tud üzenetet küldeni. Az ilyen feladatok megoldására alkalmas fejlesztői környezetekben lehetőségünk van olyan erőforráskérésre, mely után a program „egyből” megtudhatja (visszatérési kódból, állapot kódból), hogy az erőforrást meg tudta szerezni, vagy sem, minden esetben futhat

tovább, s eljárhat valahogy. Programozhatjuk úgy is, hogy valahányszor (de nem végtelenszer) kísérelje meg újra megszerezni a kívánt erőforrást, ha sikerül akkor mehet tovább, ha nem akkor üzenetet küldhet a felhasználónak, és az ő döntése szerint jár el a továbbiakban. Természetesen várakozásokat is építhetünk a programba. E módszerrel egyrészt elkerüljük a holtpontot, másrészt elkerüljük a nagyon hosszú és üzenet nélküli várakozásokat, és elkerülhetjük a program félbemaradásából fakadó gondokat is, mert a programunkat megírhatjuk úgy, hogy sikertelen erőforrásmegszerzési kísérletek miatti befejeződés előtt a szükséges tevékenységeket végezze el. A módszer számos előnye mellett egyetlen hátránya, hogy a program megírásakor kell nagyobb gonddal eljárnunk, a program bonyolultabb, tehát több hibalehetőséget is jelent, elkészítése nagyobb időt vesz igénybe. Párbeszédes programok holtpontelkerülésére ma ezt a módszert tartjuk a legjobbnak.

# *10. KOMMUNIKÁCIÓ*

Az operációs rendszer felügyeli-irányítja a számítógéprendszer munkáját. Az esetek egy részében ez minden emberi beavatkozás nélkül megfelelő lehet. Sok olyan helyzet is adódhat azonban amikor emberi beavatkozás szükséges (pl. kifogyott a papír, stb.), illetve az operációs rendszer számos lehetséges működési algoritmusa közül mi akarjuk kiválasztani a pillanatnyi alkalmazási körülményekhez legalkalmasabbat. Ezekhez szükségünk van arra, hogy információkat nyerhessünk a rendszer pillanatnyi állapotáról, és parancsokat adhassunk a kívánt működési módokra. Amikor batch módon akarunk programokat futtatni, akkor a jobok összeállításával közöljük a rendszerrel, hogy mit várunk el.

## 10.1. Az operátori parancsok

Azt a személyt, aki az operációs rendszert kezeli, irányítja számítógép-kezelőnek, vagy operátornak szokás nevezni. Az operátor feladata a számítógép, s vele az operációs rendszer működtetése, a működés során fellépő igények kiszolgálása, az előálló helyzetek kezelése. Az operátor az operációs rendszerrel a rendszer-konzol használatával párbeszédes jelleggel kommunikál. Az operátori parancsokat legalább két csoportba sorolhatjuk, a csak tájékozódást szolgáló informatív parancsok, és a működést befolyásoló parancsok. A kommunikációt nemcsak az operátor kezdeményezheti, hanem az operációs rendszer – mintegy „magától" – rendszerüzeneteket küld, ha közölni valója van.

## 10.2. Parancsnyelvek

A programozó munkája során használja az operációs rendszer szolgáltatásait (is). Az operátori parancsok (rendszere és jogosultsága válogatja mekkora) részét a programozó is használhatja. Ha a programozó batch jobot állít össze akkor az operációs rendszer számára írja le, hogy mely programokat, milyen sorrendben, milyen paraméterekkel, milyen adatokkal, milyen körülmények között kell majd futtatni. Azt a parancskészletet, melyből a jobokat összeállítja az IBM mainframeeken Job Controll Language-nek (azaz munkavezérlő nyelvnek), röviden **JCL**-nek nevezzük. A PC DOS-ban, a Widows különböző verzióiban, az OS/2-ben e célra a **batch fájlok** alkalmasak. A PC-s OS/2, a középgépes OS/400 és a nagygépes OS/390 rendszerekben a lényegében egységes **REXX** (Restructured EXtended eXecutor) nyelv alkalmas az operációs rendszer számára munkavezérlő utasítások leírására. A WindowsXP-hez külön installálható volt, a Windows7-től a Windowsoknak integráns része a **PowerShell**, ami a batch programozást nagyon sok feladat elvégzésére alkalmassá tette. A hasonló parancsfájlokat a Linuxokban shell scripteknek nevezik.

### 10.3. A felhasználó kommunikációja

A végfelhasználó rendesen valamely felhasználói programon keresztül használja a számítógépet. Így általában a felhasználói programmal kommunikál. Nem lehetetlen azonban az sem, hogy bizonyos (rendszerint elég szűk) JCL, REXX, vagy operátori parancsok használatát is – mert szükséges lehet – megengedik a végfelhasználónak is.

### 10.4. A rendszerprogramozó parancsai

Az operációs rendszer rendszergazdáját main-frame-eken rendszerprogramozónak hívják. Feladata az operációs rendszer üzembe helyezése (generálása, installálása), a „hangolása", a módosítása, és programozása. A rendszerprogramozó általában arról gondoskodik, hogy az operációs rendszer az alkalmazási környezet igényeihez legjobban illeszkedjen. Az operációs rendszer teljes parancskészletét csak a rendszerprogramozó használja. Ő tudja korlátozni, hogy az operátor, a programozó és a felhasználó mely parancsokat (mely paraméterezéssel) adhat ki. Rendszerint az operátor sem használhatja a teljes parancskészletet. Például az account (elszámolási információ) fájlok, a log (naplózás) fájlok, a password (jelszó) állományok, fontos rendszerállományok kezelését nem szokták megengedni, csak a működéshez és üzembiztonsághoz feltétlen szükséges legszűkebb körnek.

A fenti feladatkörök és a hozzátartozó jogok, használható parancskészletek a nagy, sokfelhasználós rendszerek esetében jól elkülönülnek. Minél kisebb rendszereket tekintünk annál inkább összeolvadnak az egyes funkciók. Szélső esetben a PC-knél előfordulhat, hogy a rendszerprogramozó, az operátor, a programozó, és a felhasználó is egy személy. Bár a tevékenységek ekkor is elkülöníthetők, sokszor nem is tudatosul, hogy éppen mely feladatkörnek megfelelően használjuk a rendszert, nem is kell feltétlen gondolunk erre. A „klasszikusan" rendszerprogramozónak nevezett ember-feladatkör megnevezésére a Windows rendszerekben a rendszergazda, a Linux rendszerekben a root felhasználó megnevezést használják. Rendszergazdai, illetve root jogokat más felhasználók is kaphatnak, ezzel lehetővé tesszük számukra a rendszerprogramozói beavatkozásokat is.

### 10.5. A kommunikáció kiszolgálása

Nem kizárólag az operációs rendszer és a kezelő/felhasználó kommunikációját kell megoldani. Az operációs rendszerektől elvárt szolgáltatás mindenféle felhasználó-program, program-program* (egy gépen futó programok között és hálózati kapcsolatok felhasználásával a különböző gépeken futó programok között is), operációs rendszer-program és operációs rendszer-operációs rendszer kommunikációk megoldása/kiszolgálása is. A kommunikáció általánosan használt megoldása üzenetek küldésével

* A program-program kommunikációt kicsit játékosan P2P-nek rövidítik a program-to-program angol megfelelő kiejtett alakjában a to igen hasonló a two (kettő) kiejtéséhez. Hasonlóan B2B-vel rövidítik az alkalmazás-alkalmazás (pl. két banki rendszer) kommunikációját (business-to-business).

történik. Az operációs rendszer üzenetküldő szolgáltatása gondoskodik az üzenetek fogadásáról, továbbításáról, ha szükséges átmeneti tárolásukról. Az üzenetközvetítést gyakran SEND (üzenetküldés) – RECEIVE (üzenetfogadás) mechanizmusként emlegetik.

Gyakori megoldás, hogy az operációs rendszer az érkezett üzeneteket a memóriában, vagy háttértárolón kialakított sor(ok)-ba helyezi, és vagy az operációs rendszer értesíti a futó programot arról, hogy számára üzenet érkezett, vagy a program kérdezheti meg az operációs rendszert arról, hogy érkezett-e üzenete. Ha a megcímzett program hajlandó az üzenetet fogadni, akkor az operációs rendszer azt „kézbesíti". Ha az üzenetet minden címzett megkapta, akkor az operációs rendszer az üzenetsorból törli az üzenetet (esetleg csak bejegyzi a kézbesítés tényét, és csak később, valamilyen ütemezésben törli az üzenetet).

## 10.6. A kommunikáció stílusa

Az operációs rendszerek szempontjából a kommunikáció technikai kérdés. Fogadni kell az üzeneteket és el kell juttatni a címzetthez. A neki szóló üzeneteken kívül, az üzenet tartalmával az operációs rendszer nem foglalkozik. Ha az üzenet felhasználónak, kezelőnek, általában, ha embernek szól, akkor viszont a stílusának is jelentősége van. Több (majdnem) katasztrófához vezető korábbi példa felhívja figyelmünket arra, hogy ha üzeneteket állítunk össze, akkor az legyen szakmailag korrekt (pontos, egyértelmű, világos) és a felhasználó kulturális szokásaihoz illeszkedően udvarias. Ez nem is olyan egyszerű, mint ránézésre gondolnánk. Különösen azért sem, mert ugyanazt a rendszert lényegében bárhol, bárki használhatja, így sokszor nem tudjuk előre, hogy milyen felhasználóhoz, milyen elvárásokhoz kell igazodnunk az üzenetek összeállításakor.

# *11. FORDÍTÓK, INTERPRETEREK, SZERKESZTŐK, ALKALMAZÁSGENERÁTOROK, SEGÉDPROGRAMOK*

Az operációs rendszereket sokféleképpen lehet értelmezni. Eddigi vizsgálódásunkban úgy tekintettük, mint az a programrendszer, amely a multiprogramozással fellépő igényeket elégíti ki. Gondoskodik a programok zavarmentes együttfuttatásáról, a hardver minél gazdaságosabb kihasználásáról. Ez az operációs rendszerek szűk értelmezése. Tágabb értelemben egy sor olyan kiszolgáló, kisegítő programot is az operációs rendszerekhez szoktak sorolni, melyek nem közvetlenül felhasználói igényeket elégítenek ki (azok az alkalmazói programok), hanem lehetővé teszik, segítik az alkalmazások kifejlesztését és használatát, vagy egyéb kényelmi szolgáltatásokat nyújtanak. A továbbiakban ezeket tekintjük át röviden.

Az e fejezetben áttekintett programokat a fizetős operációs rendszerekkel egyre kevésbé kapjuk meg, hanem általában külön áron beszerezhető, önálló termékként forgalmazott szoftverek. A nem fizetős környezetben (például Linuxok) gyakran az operációsrendszer-csomagokkal együtt terjesztik, bár nem „mindent", hanem a csomag összeállítója által fontosnak, hasznosnak vélt szoftvereket válogatja be a csomagba.

## 11.1. Fordító programok

A tárolt programozásnak megfelelően a programok – a memória és a processzor szerkezetéhez illeszkedve – bináris kódok alakjában kerülnek a memóriába, s ekkor képes a program utasításait a processzor végrehajtani, azaz a programot működtetni. Ez első ránézésre azt jelentené, hogy a programokat bináris kódsorozat formájában kell elkészíteni. Egy ilyen program következőképpen nézhet ki:

```
01000001111100000000000000000000010010001100000000000000000000010
01000001000100001111000111100000010100000001000000000000001001000
```

vagy ugyanez a már „sokkal olvashatóbb" hexadecimális alakba átírva (és folytatva):

```
41F0000048C000024110F1E0501000489C00C0004770F1F8 ...
```

(ez a *3.1. fejezet*ben bemutatott program eleje)

Ebből a rövid mintából is érzékelhető, hogy ezen a módon programozni nagyon keserves, rendkívül nehéz. A program bináris (oktális, hexadecimális) alakja áttekinthetetlen, nagyon nehezen olvasható, nagyon nehezen módosítható. Ez a programozó nagy igénybevételén túl azt is jelenti, hogy a felhasználói rendszerek (ami a számítógép-alkalmazás célja és haszna) nagyon lassan készülnének el, még annál is nehezebben módosíthatók, és nagyon speciális szakértelmet igényelnének.

### 11.1.1. ASSEMBLERek

A gépi kódú programozás nehézségei miatt már nagyon korán felvetődött az az ötlet, hogy az utasítások gépi kódja és az operandusok címzése helyett használjunk jobban megjegyezhető, olvasható, áttekinthető úgynevezett mnemonikus (memorizálható, megjegyezhető) kódokat, és szimbolikus adatneveket. Egy így megírt program természetesen egy az egyben nem végrehajtható, de nem túl bonyolult programmal „lefordítható" gépi kódú programmá. Ebben az elképzelésben egy-egy tényleges gépi utasításnak egy-egy mnemonikus utasítás felel meg. Az ilyen programozási nyelvet nevezzük **ASSEMBLY nyelv**nek, a fordítóprogramját pedig **ASSEMBLER**-nek. Az ASSEMBLER bemenete az ASSEMBLY nyelvű („forrás") program (a *3.1. fejezet*ben látunk példát IBM main-frame gépre írt ASSEMBLY nyelvű forrásprogramra), kimenete pedig a lefordított, gépi kódú program. Az ASSEMBLY nyelvek ma is népszerűek. Speciális szakismereteket kíván ugyan az ASSEMBLY programozás, meglehetősen hosszú is (utasításszámát tekintve) egy ilyen program, de megadja a programozónak az adott számítógép teljes és optimális kihasználási lehetőségét. Népszerűségük miatt ezek a nyelvek is fejlődtek, kényelmesebb programozási lehetőségeket is biztosítanak (makrótechnika). A makrók megjelenésével ma már nem feltétlen igaz az egy ASSEMBLY utasítás – egy gépi utasítás megfeleltetés, de a nyelv jellege változatlan. A gépi utasításokkal való megfeleltetés miatt szokás ezeket a nyelveket alacsony szintű programozási nyelveknek nevezni.

### 11.1.2. Magasszintű programozási nyelvek

A fordítóprogramok használatának ötlete azonban az ASSEMBLERnél sokkal nagyobb lehetőséget is jelent. Tág terét adja a szinte tetszőleges nyelvek megalkotásának és tényleges felhasználásának. Ezzel lehetővé válik egyrészt az alkalmazási terület igényeihez igazodó, másrészt a hétköznapi emberi beszélt nyelvekhez hasonlító programozási nyelvek (és fordító programjaik) megalkotása. Ezzel sokkal gyorsabbá és hatékonnyá vált a programozási munka, nem feltétlenül kíván sok speciális tudást, „emberközelibb" lett. A magas szintű nyelven írt program nem kötődik egyetlen konkrét számítógép típushoz, hordozható (portábilis) lesz. Ugyanazon forrásprogram gépi kódú alakja természetesen géptípusonként és operációs rendszerenként más-más lehet, de ennek elkészítése a fordítóprogramok feladata, a változatlan alakú forrásprogramból. Az alkalmazás és/vagy emberközelibb programozási nyelveket **magas szintű nyelv**eknek, fordítóprogramjaikat **compiler**-eknek nevezik.

A programozási nyelvek esetében az alacsony szintű – magas szintű megkülönböztetés általában egyértelmű, de néhány nyelv a kettő közti átmenetet képez, mert vannak egyik és másik csoportba soloható lehetőségei, megoldásai is.

## 11.2. A szerkesztés

A fordítóprogramok az input forrásprogramban leírt utasításoknak megfelelő gépi utasítássorozatot hozzák létre kimenetként. Szokásos programozási módszer a szubrutinok használata. Ezzel az egyforma (vagy nagyon hasonló) utasítássorozatnak a többszöri leírását kerülhetjük el. Ezt a filozófiát a fordítóprogramok akkor is alkalmazzák, amikor azt mi esetleg nem is érzékeljük. Egy-egy magas szintű nyelven megírt utasításnak a gépi kódban egy egész utasítássorozat felel-

het meg. Ha a fordító minden egyes alkalommal a kimeneten létrehozná a megfelelő gépi kódú utasítássorozatot, akkor az esetleg teljesen egyforma utasítássorozat az eredményül kapott gépi kódú programban igen sokszor előfordulna. Ezzel például annak a programnak a mérete jelentősen megnőne. Nem ezt a módszert alkalmazzák, hanem a fordítókhoz „gyári" rutinok sorozatát mellékelik. A fordítóprogram a lefordított gépi kódú utasítássorozatban pedig ezen „kész" modulokat meghívó utasításokat helyez el. Így a lefordított programban nincsenek elkerülhető (ebben az értelemben felesleges) ismétlődések. Megtehetnék, hogy maga a fordító helyezze el (például a program végén ezen rutinok egy-egy példányát), de többnyire nem így járnak el. Ugyanis nagyobb programokat, programrendszereket nem egyetlen programban írunk meg, hanem a bonyolultabb feladatokat egyszerűbb részekre, modulokra bontjuk.

A modulokat önállóan programozzuk, fordítjuk, önállóan teszteljük és végül egyesítjük egy futtatható programmá. Ha a fordításkor minden modulban elhelyezné a fordító a szükséges „kész" rutinokat, akkor a teljes rendszerben megint csak sokszoros – és így felesleges – ismétlődés keletkezne. Ezért a fordítóprogramok rendszerint nem a ténylegesen futtatható programot hozzák létre, hanem olyan programszövegeket, melyek más, külön lefordított rutinokat hívnak meg. Ezen rutinokból ténylegesen működtethető programot a szerkesztőprogrammal tudunk összeállítatni.

A **szerkesztőprogramok*** feladata a fordítóprogramok kimenetéből a rutinhivatkozások felhasználásával futtatható programot létrehozni.

Megtörténhet, hogy a szerkesztőprogram sem „teljes" önállóan futtatható programot hoz létre. Egy-egy konkrét alkalmazási környezetben ugyanis előfordulhat, hogy egyszerre sok programot szükséges működtetni, melyek ugyanazon rutinokat is használni kívánják. Ez megint csak nagy és bizonyos értelemben felesleges memóriaigényt támaszt. A sok program által használt rutinokból célszerű egy példányt a memóriában elhelyezni, és azt minden igénylő számára hozzáférhetővé tenni. Ha sok program ugyanazon rutint (rutinkészletet) használja, akkor vélhetően gyakran van rá szükség, s így általában rezidens módon tartjuk (virtuális memóriakezelés esetében akár a nem lapozott memóriaterületen) a memóriában. Tipikus például az adatbáziskezelő-rendszerek esete. Az adatbázist használó felhasználói programok ugyanazon rutinkészletet (az adatbáziskezelőt) használva férnek az adatokhoz, és időben egyszerre nagyon sok ilyen program működhet, érzékelhető tehát a helytakarékosság jelentősége. További előny, hogy, ha ténylegesen ugyanazt a programot (s nem annak külön-külön példányait) használják a közös feladatra, akkor egyszerűbb a párhuzamos felhasználások közötti koordináció, oly módon, hogy ez a felhasználói program programozásán semmit nem bonyolít.

Azt, hogy ilyen, csak futásidőben teljessé váló programszerkesztést használ egy rendszer, arról is észrevesszük, hogy a felhasználói program önmagában nem képes működni, hanem indítása előtt a közös rutinkészletet hozzáférhetővé kell tennünk. Ezt néha az operációs rendszer paraméterezésével, hangolásával érjük el (megadva, hogy mely rutinokat kell

* Az itt tárgyalt szerkesztőprogramot, megkülönböztetendő a szövegszerkesztőktől, a képszerkesztőktől, és általában minden más szerkesztőprogramtól, precízebben **kapcsolatszerkesztőnek (linkage editor)** szokás nevezni.

rezidenssé tenni); máskor el kell indítani valamilyen, pl. az adatbáziskezelő programot (ami valójában a közös rutinkészlet elindítását, hozzáférhetővé tételét is jelentheti). A nem gyakran használt, de mégis e módszerrel használható rutinokat nem kell feltétlen rezidensen betölteni a memóriába.

Az ilyen időben egyszerre többszörösen használható programokat ezen felhasználás sajátosságainak megfelelően kell megírni. Nem szabad például az egyéb programozásban szokásosan használt belső (lokális) változókat ugyanúgy használni, hiszen a többszörös felhasználás során ezek felülíródnak, így mivel egyik futás megelőzheti a másikat, a másik változói értékeit felülírná. Az ilyen rutinokat, tekintettel az időben egybeeső többszörös felhasználásukra, igen gondos, körültekintő módon kell megírni. Az IBM PL1 fordítója azon ritka kivételek közé tartozik, mely különösebb programozói specialitások nélkül képes ilyen kódot is létrehozni.

Az IBM terminológiájában a most tárgyalt rutinoknak két típusát különbözteti meg: **reusable** programnak nevezi az újra felhasználhatót. Ezt időben egyszerre egy példányban használja, de rezidensen a memóriában tarthatja, s igény szerint újra és újra felhasználhatjuk; programozásában csak arra kell tekintettel lennünk, hogy a programban használt változók kezdeti értékeit minden induláskor beállítsuk. Másik típus a **reentrant** program, melyet időben egyszerre tetszőleges sok program meghívhat, s egy példánya sokszorosan működik. Programozása bonyolultabb, az adatait futásonként elkülönített memóriaterületeken (melyet minden felhasználásában az operációs rendszertől kér, megállása előtt pedig felszabadít) helyezi el, biztosítva a futások egymástól független végrehajtását. A Windows környezetben ilyenek például a .DLL, a Linux környezetben a .so rutinok.

A szerkesztés lehetővé teszi azt is, hogy nagy programok, programrendszerek egy-egy rutinját a többi ismerete nélkül is kicserélhessük, s futtatható példányt hozhassunk létre. A nagyobb rendszerek szállítói gyakran élnek ezzel a lehetőséggel, így a felhasználó által programozható (user exit routin) részeket a felhasználó saját igényei szerint átalakíthatja, s a rendszer egészét nem kell forrásban rendelkezésére bocsátani, mégis teljes működtethető rendszert tud előállítani.

## 11.3. Alkalmazásgenerátor

A felhasználóhoz való közelítés a programozási nyelvekkel nem fejeződött be. Létrehoztak olyan programrendszereket, melyek – az alkalmazáshoz illeszkedő séma szerint – mintegy kikérdezik a felhasználót arról, hogy milyen igényei vannak, és a válaszok eredményeként a felhasználó elől „eltitkolva" létrehozzák az igényeit kielégítő programot. Az erre alkalmas programrendszereket **alkalmazásgenerátor**oknak nevezik. A velük létrehozott programok sokszor nem olyan jók, mintha azt egy képzett, gyakorlott programozó írta volna meg. A program természetesen helyesen működik, csak esetleg nem a feladathoz illeszkedő legalkalmasabb algoritmussal, programozástechnikai megoldásokkal. Más szempontból a felhasználó szemével nézve előny lehet, hogy a program elkészítése nem igényelt tőle programozói szakismeretet, sem programozót. Nem kell a feladatot a (tervezőnek,) programozónak elmagyarázni, a program „gyorsan" elkészül. Főleg az „ad hoc" igényeket kielégítő, esetleg csak egyszer szükséges programok elkészítésénél ezek fontosabb szempontokká válhatnak, mint a program gyorsasága, szakmailag is gondos kivitelezése.

## 11.4. Programgenerátorok

Az alkalmazói programokban a felhasználók sok többé-kevésbé hasonló megoldást igényelnek. Ezeknek a programozói megvalósítása is természetesen hasonló, de nem ritkán bonyolult, fáradságos munka, emiatt ráadásul időt rabló és hibalehetőségeket is hordoz magában. Különösen a grafikus felületek általánossá válásával igen sokszor alkalmazunk ablakokat, csúszkákat, különböző menüket és számos hasonló építőelemet. Ezek programmal való megvalósítása hosszadalmas, és az alkalmazási algoritmustól (a program feladatától) alapvetően eltérő. A felhasználói rendszerek építésének megkönnyítését szolgálják azok a megoldások, melyeket időnként 4GL (4. generációs nyelveknek), máskor visual eszközöknek (vizuális megoldásoknak) is szoktak nevezni. Ilyenek a Delphi, a VisualBasic, a VisualC és más programfejlesztői környezetek. Ezek valójában programgenerátorok. Lehetővé teszik, hogy a program megjelenési felületét minél kényelmesebben megtervezhessük, s oda tipizált, standard elemeket (ilyen-olyan dobozokat, menüket, gombokat stb.) elhelyezhessünk. Ezen objektumokhoz algoritmusokat rendelhetünk, végül összeállíthatjuk a teljes alkalmazói programot. Ezen programgenerátorok elkészítik a megfelelő magas szintű programot, amelybe ha még szükséges „kézzel" is beavatkozhatunk. Hatékonyabb, kényelmesebb programfejlesztő eszközök, mint ha mindent magunknak „kézzel" kellene programoznunk. A programgenerátor hasonlóságot mutat az alkalmazásgenerátorral. A lényeges különbség az, hogy az alkalmazásgenerátor alkalmazói-terület specifikus, a programgenerátor pedig programozási eszköz specifikus, de általában olyan sokféle lehetőséget ad, hogy az ténylegesen nem korlátozza felhasználhatóságát.

## 11.5. Az interpreterek

A programfejlesztés eszközeiként jöttek létre a parancsértelmezők, az **interpreterek**. Az interpreterek utasításokat hajtanak végre. Rendszerint párbeszédesen is alkalmazhatók. Az utasássorozat előre is megadható, ekkor programot készítünk. Az interpreter elolvas, elemez, értelmez egy utasítást és ha azt megfelelőnek találta, akkor (egy rutinjával) végrehajtja. Sok programozási nyelvhez készült interpreter is. Minthogy az interpreter nem tudja előre, hogy mely utasításokat, mi módon fogunk használni, így fel kell készülnie a nyelv összes utasításai minden lehetséges formában való alkalmazására. Egy általános programban nem szoktuk (mert nem indokolt) egy nyelv összes lehetőségeit kihasználni. Így az interpreter rutinjai szinte mindig „okosabbak" (és ezzel együtt nagyobbak és lassúbbak) az éppen feltétlen szükségesnél. Az interpreter minden egyes utasítás formai elemzését minden utasítás végrehajtása előtt elvégzi. A fordítóprogram csak fordításkor végez ellenőrzéseket, s ha jónak találta, akkor futáskor ezzel már nem múlik el idő. A fordítással (és szerkesztéssel) létrehozott futtatható program csak az éppen szükséges utasításokat tartalmazza és hajtja végre, így helyigénye kisebb, működése gyorsabb az interpreteres végrehajtásnál. Az interpreterek viszont a nyomkövetésre, általában a programfejlesztéshez nyújtanak az átlagos fordító programoknál több lehetőséget. Rendszeresen futtatandó programot mindenesetre nem célszerű minduntalan interpreterrel végrehajtatni, hanem a már kész „jó" programot le kell fordítani gépi kódúra (és persze meg kell szerkeszteni).

### 11.6. A JAVA módszere

Megjelentek a fordítás és az interpreteres program futtatás előnyeit kihasználni igyekvő vegyes megoldások is.

A fordítás „gondja" az, hogy a keletkezett gépi kódú program csak egyetlen géptípuson, s azon belül is esetleg csak egyetlen operációs rendszerben végrehajtható. Az interpreteres futtatás fő gondja, hogy a program minden utasítását végrehajtás előtt elemezni kényszerül (hogy hibátlanul írtuk-e le). Ez oda vezet, hogy ha egy utasítás cikluson (ciklusokon) belül van, akkor az interpreter azt a program futása közben igen sokszor elemezi (mert valóban, esetleg meg is változtathattuk), s így a futásidő jelentősen megnő.

A JAVA nyelvben például azt a megoldást alkalmazzák, hogy a fordítás során nem gépi kódú, hanem egy – géptípushoz nem kötött, egységes – köztes kódra fordítja a forrásprogramot, s ezt a köztes kódot egy – a magyar fordításokban sokszor JAVA motornak (néhol kissé félrevezetően Java virtuális gépnek - JVM) nevezett, valójában – interpreter program futtatja. Ezzel mindkét fentebb tárgyalt problémát kiküszöbölték. A fordítás eredménye géptől, operációs rendszertől nem függő egységes kód,* a futtatáskor ugyan a köztes kódot értelmező interpreterre van szükség, de annak már nem kell az eredeti forráskóddal „bíbelődnie" , biztos lehet abban, hogy csak a szükséges számú és értelmes utasítást kell végrehajtania, így az egyszerű interpreternél lényegesen gyorsabb futás érhető el. Különösen így van ez amiatt, hogy a JAVA fordítók elemezik és optimalizálják a köztes kódot. JAVA interpretereket („motorokat") szinte minden géptípusra, operációs rendszerre készítettek; Windows, OS/2, Linux, UNIX(ok), NetWare, MVS-re már elkészültek, népszerűségük miatt a jövőben feltehetően minden közkeletűbb rendszerben meg fognak jelenni.

Hasonló megoldással él az ActiveX és a .NET, amiket a Microsoft a JAVA vetélytársául szánt. Legalábbis e jegyzet írásakor úgy tűnik, hogy az ActiveX/.NET „csak" Microsoft által perspektivikusnak ítélt megoldás, amíg szinte az összes többi, e kérdésben meghatározó cég a JAVA jövőjében hisz.

Jóllehet a JAVA interpreter hatékonyabb az általános interpretereknél, a JAVA programok futásidejét mégiscsak növeli. A JAVA 5-ös verziójától ezen azzal igyekeznek segíteni, hogy a program futtatása előtt a köztes kódot gépi kódú programmá fordítják.

### 11.7. Segédprogramok

A most áttekintett – alapvetően az alkalmazások fejlesztését segítő – programokon kívül az operációs rendszerek számos olyan segédprogramot biztosítanak, melyeket a felhasználói rendszerek kialakításakor használhatunk (például rendező, válogató, listázó, tablózó programok), illetve a számítógép-alkalmazásban szükséges kiszolgáló funkciókat biztosítanak. Ilyenek

* Nem kell az eredeti forráskódot odaadni másoknak. A programokat szerzői jog védi. Üzleti okból, és a módosítás lehetőségének kizárása miatt is, a forrásprogramot, annak tulajdonosa nem szívesen adja oda másoknak.
A köztes kód átadásával az még „bármilyen" rendszerben futtatható, de illetéktelenül, könnyen nem módosítható. A hálózat robbanásszerű terjedésével a JAVA-szerű megoldásnak igen nagy jelentősége van, mert nem is kell tudni, hogy az alkalmazó éppen milyen rendszert használ, mégis egyetlen köztes kóddal igen sokféle rendszerben működőképes programot tudunk készíteni.

a mentő, archiváló, visszatöltő, hibafelderítő (és részben elhárító), perifériakezelő, könyvtár, katalógus, fájlkezelő, tömörítő, titkosító, hálózatkezelő stb. programok. Régebben külön hangsúlyt kaptak a rendező programok. A számítógép-alkalmazásban nagyon általános az adathalmazok kezelése. Amikor nem egyetlen adatot kell papíron, vagy képernyőn megjeleníteni, akkor azokat nem szabad „ömlesztve" megjeleníteni, mert egy rendezetlen lista gyakorlatilag semmire sem jó. (Mit is szólnánk egy semmi módon nem rendezett telefonkönyvre, vagy ha egy térkép névmutatója rendezetlen lenne stb. ezek a felhasználási célnak nem felelnek meg.) Így az alkalmazói rendszerek elkészítése során a rendezés tipikus, igen gyakran előforduló feladat. Azért fontos tudnunk, hogy a rendezésre kész programok vannak, mert ezek nagyon jó rendező algoritmusokat használnak. (Érdekességként megemlíthetjük, hogy a rendezőprogramok a szoftvergyártók régi versenyterülete. Minduntalan javítják, gyorsítják, szolgáltatásaikban bővítik rendezőprogramjaikat.) Azt mondhatjuk, hogy nem szabad saját rendezőprogramrészeket a programjainkba illeszteni, mert szinte biztos, hogy az operációs rendszerben, illetve a fejlesztői rendszerekben rendelkezésünkre álló rendezőprogramnál jobbat úgysem tudunk írni. A jobb rendezőprogramok tartalmaznak válogatási és listázási lehetőségeket is. (Az oldalszámozás módjától, a fejléc megadásán túl az összegfokozatok képzése, és általában a listákkal kapcsolatos igények kielégítésére képesek.) így nagy valószínűséggel állíthatjuk, hogy az adatállományok listázásával kapcsolatos feladatok túlnyomó többségére nem kell önálló programot készíteni, mert a rendezőprogram alkalmas paraméterezésével e feladatokat ellátja.

# *12. OPERÁCIÓS RENDSZEREKET KISZOLGÁLÓ OPERÁCIÓS RENDSZEREK*

Eddigi tárgyalásunkban az operációs rendszerek feladata az alkalmazási programok igényeinek kiszolgálása. A felhasználók esetenként igénylik, a nagykapacitású számítógépek pedig lehetővé teszik, hogy egy gépen egyszerre több operációs rendszert is működtessünk. Az operációs rendszerek fejlődésénél láttuk, hogy ez a fejlődés tanulmányozható úgy is, mint az igény jelentkezése és kielégítésének folyamata. Ezt az igényt a virtuális számítógépet biztosító operációs rendszerek oldják meg, bővebben a *12.1. fejezet*ben foglalkozunk velük.

A számítógép-alkalmazók részéről felmerülő másik igény, hogy bizonyos drága, vagy egy-egy alkalmazó által ritkábban használt perifériát közösen használhassanak. Ez azt is jelenti, hogy a felhasználók számítógépei és a közösen használni kívánt berendezések között kapcsolatot kell létrehozni. Kevés számú és egyszerűen kezelhető kapcsolatokra használhatunk célberendezéseket (például néhány PC közös nyomtató használatának megteremtésére vásárolhatók nyomtatóelosztók), közös erőforrások használata azonban bonyolultabb helyzeteket értékelni és döntéseket hozni tudó feladat. Szokásos megoldása a számítógépek hálózatba kötése, és hálózati operációs rendszerek használata (lásd a *12.2. fejezet*ben). A számítógép-hálózatok feladataival és az alkalmazott megoldásokkal e jegyzetben nem foglalkozunk, csakis az operációs rendszerek szemszögéből vizsgáljuk a hálózatokat.

## 12.1. A virtuális gép

Eddigi megközelítésünkben az operációs rendszer a hardvert közvetlenül működtető programrendszer, amely minden más program zavartalan futtatásáról gondoskodik. Többféle operációs rendszer terjedt el széles körben. Minthogy a felhasználói programok az operációs rendszerek szolgáltatásait veszik igénybe, így többé-kevésbé az operációs rendszerekhez kötődnek. Nagyon gyakran előfordulhat tehát az, hogy találunk egy, a mi felhasználói (és/vagy programozói) igényünknek nagyon megfelelő programot, de az más operációsrendszer-környezetet igényel, mint amit eddig használtunk. Természetesen az eddig használt rendszerre is szükségünk van még, hiszen ebben is számos hasznos programunk működhet.

Az operációs rendszerek – mint a hardverhez legközelebb álló programok – a számítógép hardverhez kötődnek. Így a talált, igényünknek megfelelő program gépünkön való futtatásának akadálya lehet egyrészt az, hogy mi nem a kiszemelt program által igényelt operációs rendszert használjuk (sok más fontos programunk működtetésére), másrészt a megkívánt operációs rendszer lehet, hogy gépünkön nem is működtethető. Ekkor egyrészt operációs rendszert, másrészt esetleg számítógépet kellene vásárolnunk. Ez bizony alapos akadálya lehet a kiszemelt program használatának. Részben e gond megoldása szülte azt az ötletet, hogy kifejlesszenek olyan operációs rendszert amely nem a felhasználói programok futtatását oldja meg, hanem operációs rendszerek működési körülményeinek megteremtésére és operációs rendszerek egy gépen való párhuzamos működtetésére alkalmas. Az ilyen operációs rendszert szuper operációs rendszernek is nevezik, annak okán, hogy a működés-szervezésben felette áll a „közönséges" operációs rendszereknek. A „szuper" jelző tehát nem valami különleges minőséget jelent, hanem azt, hogy ez egy operációs rendszereket *működtető operációs rendszer.*

A szuper operációs rendszer szolgáltatása a **virtuális gép**. A felhasználók a szuper operációs rendszerrel párbeszédet folytatva megadhatják, hogy milyen számítógépen (mekkora memóriával, és milyen című, típusú, méretű stb. perifériakészlettel) kívánnak dolgozni, és, hogy ebben a számítógépben milyen „közönséges" operációs rendszert kívánnak elindítani. A szuper operációs rendszer a ténylegesen meglévő erőforrásokat (memória, perifériák) az igény szerint osztja el a virtuális gépek között. Ennek kapcsán, ha szükséges, a perifériák címét és egyéb jellemzőit is szimulálja. (Például szinte bármilyen típusú és méretű mágneslemezeket kérhetünk a virtuális géphez, a valós mágneslemezeken szimulálni fogja az igényelteket.) A virtuális gépekben (a részükre kiutalt memóriaterületen) elindítja a kívánt operációs rendszert. A valós számítógép felhasználói tehát esetleg mind más-más operációs rendszerrel dolgozhatnak időben egyszerre, fizikailag ugyanazon a számítógépen.

**Operációs rendszer és felhasználói program kapcsolata**. A felhasználói programok futásuk közben az operációs rendszerektől szolgáltatásokat kérnek. Ha a felhasználói programnak valamilyen operációsrendszer-szolgáltatásra van szüksége, akkor ezt megszakítással jelzi, a megszakítással az operációs rendszer kapja meg a vezérlést, s normális esetben kiszolgálja a kérést. A szuper operációs rendszerrel működő számítógépen a (más-más operációs rendszer szolgáltatásaira számító) felhasználói programok kiszolgálása céljából arról kell gondoskodnunk, hogy a keletkezett megszakításokat mindig az az operációs rendszer dolgozza fel, amelyik illetőségi körébe tartozik. A szuper operációs rendszer a megszakításokkal kapcsolatban pontosan ezt a feladatot oldja meg. Mivel a megszakításrendszer a hardverhez kötődik, így a megszakítások természetesen (a memória előre meghatározott, fix területén tárolt címek használatával) a szuper operációs rendszer megszakítási rutinjához kerülnek. Ez általában semmit nem csinál a felhasználói programoktól és a perifériáktól érkező megszakításokkal, hanem a virtuális gépeket leíró táblázatában megkeresi, hogy melyik „közönséges" operációs rendszerhez tartozik a jelenleg érkezett megszakítás, és annak a megszakítási rutinját hozza működésbe. Ezáltal a megszakítást ugyanúgy az illető „közönséges" operációs rendszer dolgozza fel, mintha a gépen egyedül az működne, vagyis a felhasználó számára ténylegesen szimulál egy általa egyedül használt gép- és operációs rendszer környezetet. A megszakítás-átadás közben a szuper operációs rendszer az esetleges cím-átszámításokat és egyéb konverziókat is elvégzi.

**Operációs rendszerek és a szuper operációs rendszer kapcsolata**. A „közönséges" operációs rendszerek ugyan normálmódban működnek, de mikor a saját operációs rendszer funkciójukat végzik, sokszor privilegizált utasításokat hajtanának végre. Virtuális gépen működve ebből is megszakítás keletkezik, és a kívánt funkciót a szuper operációs rendszer hajtja végre. Ezáltal érhető el az, hogy az operációs rendszerek – melyek rendes körülmények között a teljes gépen mindenre képesek – a szuper operációs rendszer felügyelete alá kerülve egymást ne zavarják, illetéktelenül sem egymás, sem a szuper operációs rendszer memória és lemez területeit ne érjék el.

**A virtuális memória használata szuper operációs rendszer alatt.** Nagyon általános, hogy az operációs rendszerek virtuális memóriát használnak. Ez oda vezetne, hogy a virtuális gép memória területe (melyet a virtuális gépen futó operációs rendszer valósként lát) már egy mágneslemezen virtuális memóriában helyezkedne el, és a virtuális gépen futó operációs rendszer is a saját virtuális memória kezelésével dolgozna. Vagyis, amikor a virtuális gép operációs rendszerében futó program egy lapját a virtuális gép virtuális memóriájából a virtuális gép valós(nak hitt, de valójában szintén mágneslemezen lévő) memóriájába töltené a virtuális gép operációs rendszere,

akkor egyik mágneslemez-területről csak egy másik mágneslemez-területre íródna át a lap, és onnan a szuper operációs rendszer virtuális memóriájából a szuper operációs rendszer töltené be a valós gép valós memóriájába. Ezt a kétszeres virtuális memóriakezelést elkerülendő, a szuper operációs rendszer módosítja a „közönséges" operációs rendszerek lapozási tevékenységét. Ezt meg tudja tenni, hiszen mint minden I/O-művelet, a lapozás is privilegizált utasításokat igényel, s már említett módon, ha a „közönséges" operációs rendszer ilyet ad ki, az a szuper operációs rendszerben megszakításhoz vezet, vagyis át tudja venni a tevékenység szabályozását.

**A szuper operációs rendszerek spool-rendszere.** A szuper operációs rendszerek a „közönségesekéhez" hasonló spool-rendszert is alkalmaznak. Ez azon kívül, hogy a virtuális gépek nyomtatásait a valós nyomtatókra szervezi, alkalmas még a virtuális gépek közötti job- és adatátvitelre is.

A szuper operációs rendszerek az említett cél elérésén (egy gépen egy időben több operációs rendszer működtetése) kívül további hasznos lehetőségeket is adnak. Egyrészt fejleszthetjük, kísérleteket végezhetünk saját operációs rendszerünkön. Ez mivel az operációs rendszer hibájához, működésképtelenségéhez is vezethet, sokfelhasználós rendszerekben máshogy szinte megengedhetetlen, ugyanis zavarná a többi felhasználót, az „éles" rendszerek működtetését. A szuper operációs rendszerben csak a mi magunk virtuális gépén futó kísérleti operációs rendszerünkkel, csak mi nem tudunk esetleg dolgozni, mindenki más a többi virtuális gépen eközben zavartalanul dolgozhat. Egy másik érdekessége a szuper operációs rendszerek használatának az, hogy akár olyan felállású virtuális gépeket is kérhetünk, amilyenek ténylegesen nincsenek is, ezekkel is kísérletezhetünk.

A virtuális gépek használata gazdaságilag is előnyös lehet. A számítógépek kapacitásának növekedésével igen gyakori, hogy olyan nagyteljesítményű gépet veszünk (mert például már nem is kapunk kisebb teljesítményűt), melyet egyedül nem használunk ki. Ha másnak is szüksége van számítógépre, akkor a virtuális gép alkalmazásával elkerülhetjük újabb (és szintén ki nem használt) gép vásárlását, hanem a meglévő gépünk szabad kapacitására alapozva, abban elindíthatunk egy (esetleg több) virtuális gépet, s a többi felhasználó ezeket használhatja.

Az IBM main-frame gépekre készített szuper operációs rendszere a VM (Virtual Machine). Ez azonkívül, hogy „közönséges" operációs rendszerek működtetésére képes, saját operációs rendszer szolgáltatásokat is ad. Így tisztán VM alatt is lehet programokat fordítani – szerkeszteni – futtatni. Saját fejlesztői környezettel (CMS – Conversational Monitor System), adatátviteli spool-rendszerrel (RSCS – Remote Spooling Communications Subsystem) és utility-készlettel rendelkezik.

Az IBM 390-es és Z sorozatú gépein a szuper operációs rendszer részben mikroprogramozott változatát alapértelmezésként adja. Ekkor a valós számítógépet úgynevezett LPAR (Logical Partition)-okra oszthatjuk, (fixen) felosztva a memóriát, (százalékosan) felosztva a processzoridőt és (esetleg részben átlapoltan) felosztva a perifériákat. Innentől kezdve minden LPAR önálló számítógépként viselkedik, mintha ténylegesen különálló gépeink lennének. Ezekben aztán önállóan különböző operációs rendszerek IPL-ezhetők (tölthetők be és indíthatók el), egymástól függetlenül felhasználhatók.

A PC-kre is kifejlesztettek többféle virtuálisgép-megoldást. Ezek egyik példája az IBM **OS/2** operációs rendszere is. Az OS/2 önálló multiprogramozott operációs rendszer, amely azonban alkalmazásként képes a PC-ken működő DOS operációs rendszerek (s ezzel a DOS-ra megírt programok) futtatására is. Az OS/2-ben virtuális DOS gép(ek) hozható(k) létre.

Másik példa a VMware, amely szoftver Windows vagy Linux alatt működve lehetővé teszi virtuális gépek létrehozását és azokban (ha nem is egészen tetszőleges, de) sokféle operációs

rendszert bootolhatunk és használhatunk. A **VMware** nemcsak virtuális gépeket, hanem ezek között virtuális hálózatot is képes biztosítani.

További példák a Microsoft által fejlesztett **Hyper-V**, ami lényegében szuper operációs rendszernek tekinthető és a Windows7-hez tartozó **Virtual PC**, amely Windows7–8–10 alatt teszi lehetővé bizonyos operációs rendszerek használatát.

Sok Linuxos környezetben használhatjuk a **user-mode-linux** megoldást, mely linuxon-linuxnak tekinthető. A VMware-hez hasonlóan itt is lehetőséget kapunk belső virtuális hálózat használatára is.

Egy különlegesebbnek tekinthető virtuálisgép-megoldás a PC-s operációs rendszerekben (Linux, Windows, Mac OS) működtethető **Hercules**, amely az IBM main-frame gépeit (IBM 370, 390 és a Z sorozatú main-frame-ek) emulálja. A különlegessége abban áll, hogy ezek utasításkészlete is eltér a PC processzorok utasításkészletétől.

A most felsorolt PC-s virtuális gépet megvalósító példák jóllehet sokszor igen hasznosak, mégsem teljes értékűek. Legfőbb veszélyük az, hogy nem egy szuper operációs rendszer szolgáltatásai, hanem egy-egy „közönséges" operációs rendszer alatt futó alkalmazások. Így ha a „fő" operációs rendszerben probléma jelenik meg (miért ne, hiszen ekkor nem csak az operációs rendszer működik, hanem bármilyen felhasználói programok is, s így minden „szokásos" hiba előfordulhat) akkor a virtuális gép működésében is (ráadásul nem is tőle származó) zavar keletkezhet. Ezt a veszélyt csökkentendő általában azt javasolják, hogy ha virtuális gépet használunk, akkor az alap operációs rendszerben más programot ne futtassunk, csak a virtuális gépeket biztosító szoftvereket.

**Szimuláció.** Meg kell különböztetni a szuper operációs rendszer által biztosított virtuális gépen történő „közönséges" operációs rendszer működtetést az operációs rendszerek szimulációjától. Amikor valódi operációs rendszer működtetésről beszélünk, az azt jelenti, hogy a „közönséges" operációs rendszerek kicserélhetőek, módosíthatóak, bármi változtatható velük kapcsolatban, és mindig ugyanúgy viselkednek, mint amikor ténylegesen egyedül működnek egy valós gépen. A szimulált operációsrendszer-funkciók ezzel szemben nem (illetve csak nagyobb nehézségek árán) változtathatók. A szimulációval egy rendszer gondoskodni igyekszik egy más rendszer szolgáltatásainak biztosításáról is, ezáltal a másik rendszerre elkészített programok működtetését ígéri. Ez eleinte többé-kevésbé igaz is (nem mindig gondoskodnak minden funkció szimulálásáról), de a „közönséges" operációs rendszer későbbi változataira már egyre kevésbé. Azt mondhatjuk, hogy a szimuláció sokkal rugalmatlanabb, így előre nem várt kellemetlen akadályokba ütközhetünk vele. Ilyen szimulációs megoldás például az egyes Linux-csomagokban előforduló wine (windows emulator*) windows-szimulációt biztosító program; a Windows XP alatti DOS stb.

* Az emuláció itt utánzást, a szimuláció pedig tettetést, színlelést jelent.

## 12.2. Hálózati operációs rendszerek

A hálózatba kötött és a felhasználó által közvetlenül használt számítógépeket általában a „szokásos" operációs rendszerekkel működtetik. Az olyan számítógépeket

melyek a hálózat többi tagja számára nyújtanak valamiféle szolgáltatásokat host-, illetve szervergépeknek, nevezzük. **Host (vagy gazda)** gépről akkor beszélünk, ha a többiek számára nyújtott szolgáltatások között felhasználói programok futtatása is „szerepel, **szerver (vagy kiszolgáló)** gépről beszélünk, ha a többiek (futó programok) számára valamilyen szolgáltatást nyújt. Ez lehet lemezhasználat (fájl- vagy diszkszerver), lehet adatbázis-szolgáltatás (adatbázisszerver), lehet név-cím szolgáltatás (névszerver, címszerver), levelezési szolgáltatás (levelező-, mailserver), Webszerver stb. Egy számítógép egyszerre host- és szerverszerepet is betölthet. A hostfeladat (programok futtatása) operációs rendszer szempontjából nem jelent semmi lényegesen újat. A szerverfeladat igen, azt kell ugyanis biztosítania a szerver gépet működtető operációs rendszernek, hogy az ő perifériáit, szolgáltatásait más gépeken futó (értelemszerűen az ott működő operációs rendszer felügyelete alatt futó) programok is használhassák. Ez más megfogalmazásban azt jelenti, hogy meg kell oldani azt, hogy a felhasználói programot működtető számítógépek operációs rendszere a szerver bizonyos perifériáit és/vagy programjait is úgy, vagy ahhoz hasonlóan használhassa, mintha sajátja lenne.

Ez természetesen a szerveren egy sor feladat megoldását igényli, hiszen időben átlapoltan egyszerre sokan jelentkezhetnek kiszolgálási igénnyel. Nos, a szervert működtető ilyen operációs rendszert szokták **hálózati operációs rendszer**nek nevezni. Erre példa a PC-hálózatokban használatos Novell NetWare operációs rendszer, az IBM Lan Server Entry operációs rendszere, a Windows több verziójának server változatai és még számosan mások. A hálózati operációs rendszerek feladata nem felhasználói programok futtatása,* hanem a többi számítógépet működtető operációs rendszerek számára perifériakezelés megoldása és/vagy kiszolgáló programok működtetése. Így beszélhetünk diszkszerverről, ennek szolgáltatása a saját mágneslemezei többiek számára elérhetővé tétele; fájlszerverről, ennek szolgáltatása a fájlok közös használatával kapcsolatos feladatok megoldása, nyomtatószerverről, mely a közös nyomtató-használatot biztosítja, hálózatokban beszélhetünk mailszerverekről, melyek levelezési feladatokat oldanak meg, címszerverek, melyek a szimbolikus megjegyezhető hálózati címekhez (mint egy telefonkönyv) megadják a tényleges hálózati címeket, s a szerverek sorát hosszan folytathatjuk (Web-szerverek, Proxy-szerverek...).

A hálózati operációs rendszerek természetesen minden olyan feladatot ellátnak, amelyek a többfelhasználós környezetekben általánosak, így a „közönséges” operációs rendszerek feladatai között már foglalkoztunk velük. Ilyenek a hozzáférés-felügyelet, jogok, korlátozások kezelése, elszámolási információk gyűjtése (esetleg feldolgozása), naplózás stb.

A „közönséges” operációs rendszer – hálózati operációs rendszer „tiszta” képet árnyaltabbá teszi, hogy az alapvetően „közönséges” operációs rendszerként épített operációs rendszerek (melyek fő feladata egy számítógépen a programok zavartalan futásáról gondoskodni) is adhatnak többkevesebb korlátozott hálózati kiszolgálást is. Lehetővé tehetik a hálózat többi gépe számára a mágnes- és CD- (DVD-) lemezeik, vagy azok egy

* Bizonyos hálózati operációs rendszerek lehetővé teszik a felhasználói programok futtatását is a hálózat kiszolgálásával egyszerre. Ekkor a hálózat kiszolgálását nem-dedikáltnak nevezzük. Általában – különösen kisebb teljesítményű gépeken működő – hálózati operációs rendszerek „csak” a hálózat kiszolgálásával foglalkoznak, ezt dedikált hálózat-kiszolgálásnak nevezzük. A kisebb teljesítményen kívül a hálózati kiszolgálás üzembiztonsága miatt is a dedikált módot szokták javasolni. Nem dedikált környezetben ugyanis egyetlen felhasználói program hibája a szerver működésében zavart okozhat, megzavarva ezzel sok más (más gépeken futó de a hálózati operációs rendszer szolgáltatását is igénybe vevő) programok működését.

részének elérését, a nyomtatójuk közös használatát, az internet-kapcsolatuk megosztását a „többiekkel". E szolgáltatások célszerűek, gazdaságosak akkor, ha egy kisebb számítógépes közösség (egy iroda, kis cég, otthoni felhasználó) néhány számítógépet használ, és szeretne mágneslemezt, CD/DVD-t, nyomtatót, internet-kapcsolatot közösen használni. Ilyen esetben, ha vállalják a nem-dedikált használatból következő kockázatot, akkor nem kényszerülnek önálló szerver(ek) és szerver-szoftverek beszerzésére.

A számítógép-hálózatok tárgyalásával e jegyzetben bővebben nem foglalkozunk, a számítógéphálózatok tanulmányozása az informatika egyik nagy, önálló területe.

# *13. AZ OPERÁCIÓS RENDSZER INSTALLÁLÁSA, HANGOLÁSA*

Az operációs rendszerek a mai számítógépeken rendesen mágneslemeze(ke)n helyezkednek el. Új mágneslemezen – a mi szempontunkból – nincs semmi. Az is előfordulhat, hogy egy mágneslemezen már van ugyan egy operációs rendszer, de most egy más operációs rendszert szeretnénk elhelyezni rajta. Természetes feladat tehát az operációs rendszerek üzembehelyezésének (installálásának) feladata. Mivel meglehet, hogy még nincs üzemszerűen működő operációs rendszerünk, így annak használatára nem számíthatunk. Ezért az operációs rendszereket úgy szállítják (mágneslemezen, mágnesszalagon, optikai lemezen stb.), hogy azok önmagukban üzembehelyezhetőek legyenek. Ennek általános megoldása az, hogy egy minimális és minden szokásos konfiguráción működni képes „starter" operációs rendszer készen van a szállított adathordozón. Ez a minimális operációs rendszer általában nagyon keveset tud, többnyire csak néhány segédprogram működtetésére képes (lemezformázó, másoló, szövegszerkesztő, assembler, legfeljebb C-fordító, szerkesztő programok), rendes, üzemszerű számítógép-használatot nem biztosít, csak az operációs rendszer üzembehelyezésére alkalmas. Ha az installációs anyag cserélhető lemezen van, akkor arról a starter rendszer egyből elindítható, ha más módon (szalag, optikai lemez) szállítják, akkor rendszerint adnak egy lemezformázó és egy lemezre letöltő stand-alone programot is, s ezekkel a starter mágneslemezre tölthető. Elérhető tehát a starter rendszer elindítása, s abban a tényleges rendszer installációját tudjuk megkezdeni. Az installáció – természetesen – operációs rendszerenként más-más módon történik. Általánosságban elmondható, hogy meg kell adnunk a használni kívánt számítógép hardver konfigurációját (ez részben automatikusan is felderíthető, de például az éppen kikapcsolt vagy más ok miatt el nem érhető perifériákra nem), és meg kell adnunk az operációs rendszer fő paramétereit. Az operációs rendszer működési paramétereit részben az installációkor, részben a rendszerek indításakor, és részben működés közben is megadhatjuk. Az operációs rendszer működési paraméterei megadását az **operációs rendszer hangolás**ának (konfigurálásának) nevezzük. Ez rendesen a rendszerprogramozó feladata, ehhez az operációs rendszer több-kevesebb segítséget is ad. A hangolással állítjuk be a konkrét alkalmazási körülményeknek legjobban megfelelő működési módot.

A hangoláshoz tartozik például a rezidenssé teendő programok meghatározása, ennek jelentős hatása van a rendszer felhasználó által érzékelt sebességére és a szabad memória méretére, de a hangoláshoz tartozik az időosztási-prioritási eljárások, az automatikus mentési, tükrözési, a jogosultság és munkaellenőrzési módok megadása, általában minden megváltoztatható paraméter értékének beállítása. Az indításkor megadható paraméterekre példa a PC-k DOS operációs rendszereiben használatos CONFIG.SYS és részben ide sorolhatjuk az AUTOEXEC.BAT állományokat. Az IBM MVS operációs rendszerben a SYS1.PARMLIB állomány összes membere rendszer-paramétereket tartalmaz. Az OS/2 paraméterállományai az OS2.INI és az OS2SYSTEM.INI. A Windows rendszerek paraméterállománya a REGISTRY.

## 13.1. Amikor nincs operációs rendszer, a stand-alone programok

Alapvetően két olyan eset(csoport) van, amikor nincs rendesen használható operációs rendszerünk. Az egyik eset a normálisnak tekinthető, előző pontban tárgyalt, amikor egy operációs

rendszert korábban nem tartalmazó mágneslemezen üzembe kell helyeznünk a kívánt operációs rendszert. A másik – sajnos szintén előforduló eset – amikor az üzemképes operációs rendszerünk megsérült. Ekkor több mindent lehet tenni, lehet, hogy egy másik lemezen van tartalék operációs rendszerünk, és átmenetileg használhatjuk azt. (Általában azért csak átmenetileg, mert a tönkrement rendszerben lehetnek olyan adatok, melyeket meg kell menteni, mind felhasználói rendszerekhez tartozó adatok, mind az operációs rendszerek account, hibaregisztrációs, naplózási, biztonsági stb. fájljait.) Többnyire arra törekszünk, hogy a lehető legkisebb veszteséggel a meghibásodott rendszert hozzuk helyre. Ehhez, és az üzembehelyezéshez is, úgynevezett **stand-alone,** azaz operációs rendszer nélkül, önmagában is futtatható programokat biztosítanak. Ezek a programok természetesen csak egyedül működnek, gondoskodniuk kell saját betöltésükről, és például a megszakítások – számukra szükséges részét (operációs rendszer nem lévén) – maguk kezelik. Ilyen programok – egyedül működve – természetesen nagyon rosszul használják ki a számítógépet, s így kizárólag a hiba elhárítására érdemes használni őket. Általában lemezinicializáló, lemez-szalag, szalag-lemez, lemez-lemez másoló, illetve lemez bizonyos területeit „kézzel" módosító programokat szoktak stand-alone formában is elkészíteni.

PC-s környezetekben tipikusan nem alkalmaznak valódi stand-alone programokat. Sokkal inkább az operációs rendszer (DOS, Windows, Linux) egyszerű, „lebutított" (de éppen ezért szinte minden környezetben működőképes) változatát helyezik el egy-egy bootolható floppyra, CD/DVD-re. Probléma esetén erről bootolva kapjuk meg a helyreállítási lehetőséget. Ez teljesen egyenértékű (sőt a bővíthetősége miatt jobb) megoldás, a „klasszikus" stand-alone programokkal. Ilyen bootolható lemez elkészítését minden olyan korrekten elkészített program felkínálja (operációs rendszerek, víruskeresők, boot-managerek stb.) melyek működési hibája esetén az operációs rendszer működésében (következésképp a teljes rendszerben) zavar keletkezhet. Ezeket a bootolható lemezeket a különböző rendszerek helyreállító-, vész-, emergency-, biztonsági-, safe-, javító-, correction- stb. lemeznek nevezik.

# *14. VIRTUÁLIS MEGOLDÁSOK*

A jegyzet korábbi fejezeteiben már találkoztunk olyan szoftver-megoldásokkal, melyek lehetővé teszik azt, hogy egy adott program olyan környezetben működhessen, amilyet a készítője és/vagy az alkalmazója kíván.

Ebben a fejezetben – a korábbiaktól részben eltérő megközelítésben – áttekintését adjuk a számítógépeken alkalmazott virtuális megoldásoknak.

Egy program működéséhez olyan környezetre van szükség, amilyenre azt (a programot) eredetileg megírták. Ha az adott program például CD-ről kíván olvasni, és mágnesszalagra óhajt írni, továbbá bizonyos utasításokat kell, hogy végrehajtson, akkor első ránézésre csak olyan körülmények között működtethető, ahol van CD, mágnesszalag és olyan processzor, mely a programozott utasítások végrehajtására képes. Ha azonban belegondolunk a program tényleges végrehajtásába, akkor azt látjuk, hogy a periféria-műveletek végrehajtását a program (egy-egy megszakítással) az operációs rendszertől kéri. A tényleges végrehajtás az operációs rendszer és esetleg a periféria-meghajtó programjai (driver) által történik, s a program csak az eredményességről, vagy eredménytelenségről értesül, de arról nem, hogy például az általa kezdeményezett, mágnesszalagra való írás valóban mágnesszalagra történt-e, vagy például mágneslemezre. A *7. fejezet*ben, a hiányzó processzorral kapcsolatban, a *12.1. fejezet*ben pedig a main-frame processzor PC-n való emulálásakor azt is láttuk már, hogy egy adott processzor által nem ismert utasítás végrehajtási kísérlete sem csak hibaállapothoz, hanem – szintén a megszakításrendszeren keresztül – az utasítás feladatának programmal való megvalósításához is vezethet.

Idézzük fel az operációs rendszerek feladatai közül a legfontosabbat (az 1.7. fejezetben fogalmaztuk meg): „a felhasználói programok működési körülményeinek biztosítása, igényeik kiszolgálása". E megfogalmazásba jól illeszkedik az is, hogy ha a felhasználói programnak valamilyen processzorra, memóriára és perifériára (ezekből állt a számítógép-modellünk) van szüksége, akkor, ha ezek fizikai valóságukban nem állnak rendelkezésre, akkor ezek funkcióit az operációs rendszer alkalmas programokkal biztosíthatja. Ilyenkor beszélünk virtuális készülékekről. A felhasználói program azt tapasztalja, hogy az utasításai, kérései végrehajtódnak, de a kívánt funkciókat a megfelelő hardver helyett szoftverek valósítják meg. A továbbiakban összefoglaljuk a virtuális megoldásokat, megemlítve néhány (biztosan nem az összes) konkrét megvalósítást.

## 14.1. Virtualitások

**Virtuális processzor.** A *7. fejezet* bevezetőjében és a *12.1 fejezet*ben foglalkoztunk vele. A megoldás lényege, hogy ha a futó program olyan utasítást kísérel meg végrehajtani, mely a processzor számára nem ismert, akkor ez megszakításhoz vezet, s ennek feldolgozásakor az operációs rendszer a kívánt utasítás funkcióját megfelelő programokkal megvalósíthatja.

Egyes rendszerekben, például az IBM nagygépes rendszereiben a virtuális processzor(ok) teljesítményét (azt, hogy melyik virtuális processzor a valós processzor teljesítményének hány százalékát kaphatja meg) is tág határok között szabadon paraméterezhetjük.

**Virtuális memória.** Az *5.3. fejezet*ben részletesen foglalkoztunk vele.

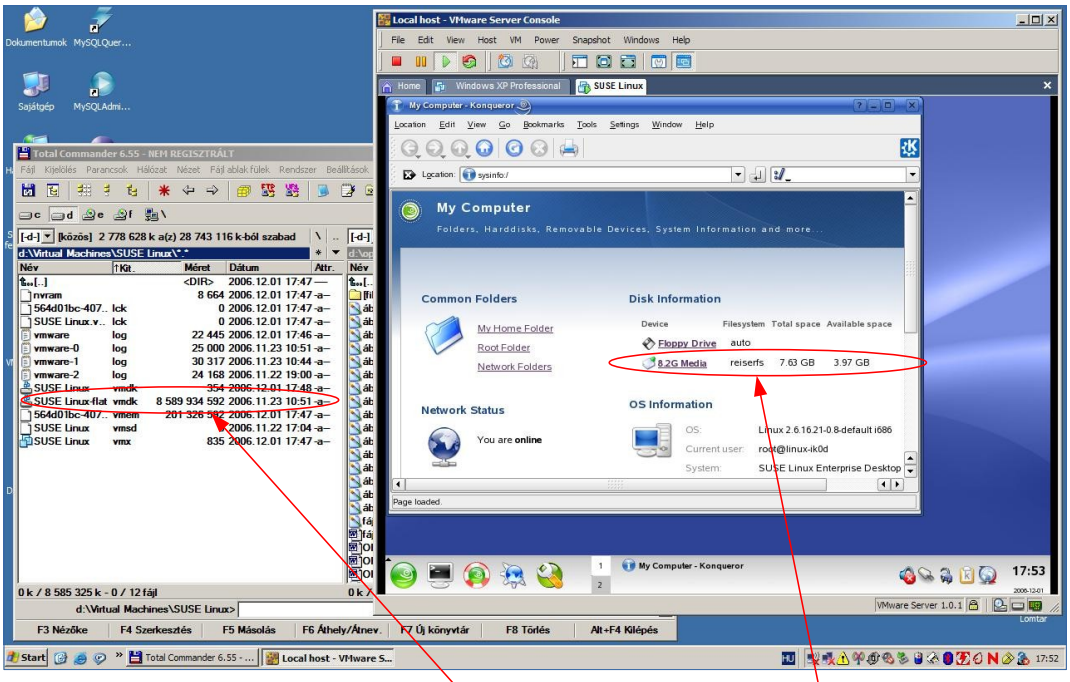
**A virtuális perifériákról.** Általánosságban a virtuális perifériák megvalósításának alapgon-

dolata az, hogy a futó programok a perifériaműveletek elvégzését egy-egy megszakítással az operációs rendszertől kérik. Így az operációs rendszernek módjában áll a tényleges periféria helyett egy-egy olyan programot működtetni, mely szimulálja a kért periféria működését. Az alábbiakban áttekintjük néhány virtuális periféria gyakrabban használt lehetőségeit.

**Virtuális mágneslemez.** Egy megközelítésben az *5.4. fejezet*ben már találkoztunk vele. Ott a gyakran memóriadiszknek, vagy RAMdiszknek nevezett megoldást vizsgáltuk, melynek lényege, hogy mágneslemez helyett memóriát használva a programok működése jelentősen felgyorsítható.

A virtuálismágneslemez-megoldások közé sorolhatjuk a különböző RAID-módszereket. Ezek alapvetően üzembiztonság-növelő módszerek, amikor több tényleges mágneslemezből olyan redundáns rendszert alakítanak ki, mely egy, esetleg két fizikai diszk tönkremenetele esetén is biztosítja minden korábban tárolt adat megőrzését. RAID-rendszerek megvalósíthatók akár szoftverrel, akár hardverrel. Sebességi meggondolások és a processzornak e feladat alóli mentesítése miatt a hardveres megoldást általában szerencsésebbnek tartják.

Más módon is találkozunk virtuális lemezekkel. Amikor egy program, vagy akár egy operációs rendszer számára biztosítunk egy általa kívánt méretűnek és a kívánt tulajdonságokkal rendelkezőnek látszó/mutatkozó területet. Ekkor lehetséges megoldás az is, hogy a valós mágneslemezt, vagy egy partícióját, vagy egy könyvtárát, vagy „csak” egy fájlját használjuk a virtuális lemez céljára. Ezt a megoldást használja például a VMware akkor amikor például egy NTFS-fájlrendszerű mágneslemezen egy .vmdk-fájlban például egy Linux-szal működő virtuális gép számára biztosít mágneslemezt, melyet a Linux önálló mágneslemeznek lát és pl. reiserfs fájlrendszerrel használ. Ilyet látunk a *14.1. ábrán*.




*14.1. ábra. A VMware által biztosított virtuális mágneslemez.*

**Virtuális CD/DVD**. A virtuális CD/DVD-használat egyik különös, de gyakran jól használható lehetősége, amikor egy CD/DVD „képfájlt”* (.ISO) tényleges CD/DVD-re való kiírás helyett a képfájlból is CD/DVD-ként tudjuk használni. Ezt a lehetőséget biztosítja például a Daemon Tools program. Ha ezen program segítségével csatlakoztatjuk (mount-oljuk) az ISO CD/DVD képfájlt, akkor amíg le nem kapcsoljuk (unmount) addig egy új CD/DVD-eszközként tudjuk használni. Ezzel időt, eszközt spórolhatunk és még futásidőben is jobban járunk.

**Virtuális nyomtató.** A „szokásos” virtuális perifériahasználaton felül a virtuálisnyomtató-megoldást használják a számítógépről történő faxolás és a pdf (portable data format) formátumú dokumentumok elkészítésére. Mindkét módszer egy-egy – különben klasszikusan kinyomtatható – anyaggal tesz valamit, továbbítja faxként, vagy készít belőle egy adott formátumú fájlt. A felhasználó (ember vagy program) számára, de magában az operációs rendszerben is nyomtatókét mutatkoznak. Pdf „nyomtatók” például az Adobe Acrobat, és a PrimoPdf programok által megvalósított virtuális nyomtatók.

**Virtuális képernyő.** A virtuális képernyő tipikus megvalósítása az ablakos technika. Egy-egy program a valós képernyő egy-egy területét használhatja, mint képernyőt. Valójában a program közvetlenül nem a képernyő területeit használja, hanem megjelenítendő anyaga memóriába (esetleg más tárolóba) kerül, s az operációs rendszer ezt a tartalmat, vagy annak egy részét jeleníti meg a képernyőn. Ha a megjelenítésben nem fér el egy ablakba szánt anyag egésze, akkor az ablak szélein megjelenő csuszkákkal szabályozható a ténylegesen képernyőn megjelenített rész. A képernyőterület pillanatnyi méretét általában nem a program, hanem a felhasználó állapítja meg, lehet akár nulla méretű, akár teljes képernyő méretű, s a kettő között bármi. (A képernyőterület átméretezését a program esetleg korlátozhatja, vagy akár tilthatja is.)

**Virtuális hálózat.** Ha egy fizikai számítógépben több virtuális gépet működtetünk, akkor a virtuális hálózat lehetőséget ad arra, hogy a virtuális gépek számára virtuális hálózatot is kialakítsunk. Ezen keresztül a gépek egymással, és más (akár valóságos) hálózatokba kötött gépekkel hálózati kommunikációt folytathatnak. Ilyen virtuális hálózati szolgáltatást biztosítanak például a VMware rendszer és a user-mode-linux.

A számítógép-hálózatokban is beszélnek virtualitásokról, például virtuális LAN (VLAN) hálózatokról. Ez a megoldás azt jelenti, hogy hálózati eszközökkel egy-egy (egymástól akár nagy távolságban lévő számítógépekből álló) számítógépcsoport számára olyan hálózathasználatot biztosítanak, mintha az adott gépek LAN hálózatban lennének egymással összeköttetésben. E témával bővebben a számítógép-hálózatok tanulmányozásában foglalkoznak.

**DUMMY vagy NULL-perifériák, -fájlok**. A virtuális periféria szolgáltatásokhoz sorolhatjuk azokat a lehetőségeket, amikor egy-egy fájl, vagy periféria nem létezik, vagy létezik ugyan, de a programot – annak módosítása nélkül – most úgy akarjuk működtetni, hogy az adott perifériával, fájllal ne dolgozzon. Ekkor

* Ez nem fényképet tartalmazó képfájl, hanem ha a CD/DVD teljes tartalmát – a tárolással mindenben megegyező formában – egy fájlba írjuk, akkor azt is képfájlnak szokták nevezni, ennek CD/DVD-re felírásával, vagy a virtuális CD/DVD segítségével az eredeti CD/DVD-tartalom válik használhatóvá.

élhetünk a DUMMY (vagy NULL) perifériaként való megadással. Az operációs rendszer ilyenkor úgy viselkedik, hogy ha a program az így megadott erőforrásra írni akar, akkor az operációs rendszer a program számára az írásműveletet sikeresnek fogja mutatni (bár tényleges írás sehova nem történik), az olvasásművelet pedig – az olvasási művelet jellegéhez igazodva –„fájl vége" vagy „nem létező rekord" állapotot fog jelezni.

Az egyszerűsített számítógép-modellünk processzorból, memóriából és perifériákból állt. Mint látjuk, ezek mindegyike lehet virtuális is, ilyenkor ezek együttesét **virtuális számítógép**nek is tekinthetjük.

A virtualitások használata egyre jobban terjed, hiszen egy adott hardver-szoftver környezetben lehetővé teszi nagyon változatos igényű felhasználói programok működtetését. Alkalmas továbbá a kapacitások igény szerinti átcsoportosítására. A virtualitások alkalmas paraméterezésével külön ráfordítás nélkül, a pillanatnyi igényekhez igazodóan, igen szélesen skálázható kapacitást biztosíthatunk a programjaink számára.

**Virtuális felhasználó.** Nemcsak a berendezések, műszaki eszközök viselkedését utánozó program készíthető, hanem írhatunk programot arra a feladatra is, hogy néhány vonatkozásban a felhasználót helyettesítse. Amikor például egy felhasználói rendszer elkészítése abba az állapotába ér, hogy a felhasználónak (esetleg nagyon sok felhasználónak egyszerre) kellene a rendszer kísérleti működtetése során sokféle akciót sokféle adattal kipróbálnia, akkor – legalábbis részben – megkímélhetjük a felhasználókat azzal, hogy írunk olyan programot, mely a vizsgálandó felhasználói rendszer számára helyettesíti a felhasználókat. Az ilyen program képes lehet helyes és hibás adatokat előállítani, és azokkal a vizsgálandó rendszert működtetni, akár a helyes és helytelen válaszokat is értelmezni, naplózni, segítve ezáltal a rendszer ellenőrzését, javítását, fejlesztését.

Természetesen a virtuális felhasználó nem egy virtuális ember, nem az ember összes funkcióit utánozza, hanem „csak" az emberi tevékenységeknek a szoftver használatával kapcsolatos részét szimulálja. Ugyanígy a virtuális nyomtató sem használ papírt (ahogy a valódi), a virtuális CD/DVD pedig nem zúg, ahogy a valódi. A valóságos dolgokat helyettesítő virtuális megoldások a valóságos dolognak csak azon funkcióit utánozzák, melyeket az adott helyzet kíván. (Ha tehát például éppen egy számítógép keltette zajokat akarjuk utánozni, akkor a virtuális CD programjába a zajkeltés képességét is programozhatjuk.)

## 14.2. Felhő

Nem szorosan tartozik az operációs rendszerek tanulmányozásához, de az operációs rendszer közreműködése szükséges a felhő használatához is. Így a felhők néhány fő jellemzőjét röviden áttekintjük. A *felhő alapú számítástechnika* (cloud computing) olyan szolgáltatáscsoport összességét jelenti, amikor a számítógép-hálózaton keresztül adattárolási és adatfeldolgozási (program futtatási) lehetőséget biztosítanak. A felhőszolgáltató elegendően nagy tároló és feldolgozó kapacitást alakít ki, és ezt bocsájtja a felhasználói rendelkezésére. A felhő néhány már korábban kialakított lehetőséget, mint például a fájlszerver, a távoli elérés (távolról történő program futtatás), a virtuális számítógépek használata, foglalja egységes „keretrendszerbe".

A felhő használatának előnyei:

– Mentesül a saját, helyi tárolókapacitás kialakításának költségétől és üzemeltetésével járó fel-

adatoktól (mentési, helyreállítási, naplózási stb. feladatok). A tárolás nagy megbízhatóságú abban az értelemben, hogy a felhőben egy adatot tipikusan több, egymástól távoli helyen is tárolnak, s így egy-egy valahol bekövetkező bármilyen helyi probléma előfordulása esetében is az adat nem vész el.
– Hasonlóan mentesül attól, hogy ha programjai működtetése nagy számítási kapacitást igényelnek, akkor ezt magának kelljen megvásárolni, működtetni. Ezt (összes velejáróival, karbantartás, javítás, szakemberigény stb.) biztosíthatja a felhő.
– Mentesül egyes szoftverek beszerzési költségétől, karbantartásuk, követésük feladataitól. A felhő szolgáltatója is biztosíthat programokat és azokat a felhasználó futtathatja a felhőben biztosított számítógépeken.
– A felhőből igénybe vett kapacitás rugalmasan változhat, igazodhat a pillanatnyi igényekhez.
– Helyfüggetlenség: az adatok és a programok a hálózat bármely pontjáról (ha azt külön valamilyen ok miatt nem gátolják) elérhetők.
Ugyanakkor a felhő használata olyan kockázatokkal is jár, mint például:
– A felhőben tárolt adatok feletti ellenőrzést elveszítjük. Nem tudjuk hol tárolódik az általunk elhelyezett adat. Ez bizonyos adatok esetében jogi problémákat is felvet, hiszen egyes esetekben el kell(ene) tudnunk számolni az adat tárolási helyéről, és annak körülményeiről.
– Semmi garancia nincs arra, hogy a tárolt adathoz illetéktelen ne tudna hozzáférni. Bár a felhőszolgáltatók ezt általában ígérik, de a helyfüggetlenséget is kihasználva, szinte bárki támadhatja is a felhőt. A kódolt, elzárt adatok megszerzésében csak szellemi kalandot látó lelkes fiataltól, a szinte korlátlan anyagi-, gépi- és szoftver-kapacitásokkal rendelkező kiberháborúzó szervezetekig sokan foglalkoznak a kibertérben elérhető információk megszerzésével. Már sokszor kellett megtapasztalni, hogy korábban biztonságosnak tartott rendszerekbe is bejutnak illetéktelenek. Érzékeny információk esetében (ilyenek a személyes, céges, banki stb. információk) ez nagy veszélye, és esetenként jogi akadálya is a nyilvános felhő használatának.
– Ha a felhőben programot is használunk, az is sok információt ad ki rólunk (melyik programot, mikor, mire használtuk).
– A felhő használója zsarolhatóvá válik. Ha az információk nem kerültek illetéktelen kézbe, akkor is előfordulhat, hogy a szolgáltató kezdetben kedvező árat kínálva magához vonz felhasználót, majd később az árat már szinte korlátlanul emelheti, mindaddig, amíg a felhasználó esetleges vesztségének értéke közelébe nem kerül.
– Az adatok törlését nem tudjuk garantálni. Bizonyos adatok törvényekben, vagy más előírásokban meghatározott módon gyűjthetők, de előírt idő után törlendők. A felhőben – hasonlóan az internethez – nem garantált, hogy a törölni szándékozott adat valóban törlődik-e, vagy csak a „normál” alkalmazás számára tűnik nem létezőnek, de valójában még létező.

A felhő elnevezést a fenti problémák miatt többen igen eltaláltnak tartják. A valóságos (meteorológiai) felhők is távolról lehetnek nagyon szépek, közelről azonban sűrű köd, amelyben nem tudjuk pontosan mi történik, ki mit tesz. Sokkal inkább balesetet szenvedhetünk benne, mint rajta kívül. Így még privát alkalmazóként is, igen erősen megfontolandó, hogy használjuk-e a felhőt, s ha igen mire; vállalati, „céges” alkalmazhatósága pedig különösen korlátozott, erősen átgondolandó.

Az összképet azért árnyalja az, hogy nemcsak **nyilvános felhők** használhatók, hanem a felhő megvalósítására kidolgozott megoldásokkal egy-egy cég – illetéktelenektől elzárt – saját, **privát felhőt** is építhet. Ez alkalmas lehet a saját adatok (cégen belüli) elérésére, a kapacitások rugalmas felhasználására. E feladatok kisebb-nagyobb része persze már korábban is megoldott volt fájlszerverekkel, távoli elérések biztosításával és hasonló technikákkal.

# *ZÁRSZÓ*

Ez a jegyzet nem a téma teljes és részletes feldolgozása. A jegyzet kifejezett célja az operációs rendszerek bevezető tantárgya előadásai követésének segítése. Az operációs rendszerek – hasonlóan az informatika számos más területéhez – igen jelentős fejlődésben vannak. Fontos, hogy a hallgatók, és majd mind szakemberek, minél naprakészebb tudás birtokába juthassanak, ezért az előadások anyagát és hozzá illeszkedve a jegyzetet is rendszeresen átdolgozom. Aki e témában, vagy bizonyos részeiben jobban el kíván mélyedni, annak figyelmébe ajánlom az irodalmi összefoglalásban szereplő művek és a szoftverek kézikönyvei tanulmányozását. A már említett ok miatt, az operációs rendszerekkel foglalkozó irodalmak is állandóan bővülnek. Az operációs rendszerekkel foglalkozó szakembereknek rendszeresen figyelemmel kell kísérniük a szakirodalmat is!

A jegyzettel kapcsolatban köszönettel fogadok minden megjegyzést, javaslatot!

Dunaújváros, 2016. március

*Dr. Buza Antal*

# *TÁRGYMUTATÓ*


.NET, 104
4GL, 103

account, 18
ActiveX, 104
adatok védelme, 18
alkalmazásgenerátor, 99
alulcsordulás, 22
ASSEMBLER, 100
ASSEMBLY, 100

állapot
    megvalósítható, 82
    megvalósíthatatlan, 82, 84
állapotregiszter, 24

B2B, 96
batch, 95
batch fájl, 95
bázisregiszteres címzés, 16
BC mód, 23
Bernstein-tétel, 79
boot, 35
Boot Record, 35

cache, 53, 69
Cache Only Memory Access, 69
CDFS, 63
címszámláló, 12
cloud, 118
COMA, 69
compiler, 100
CPU, 67
CS regiszter, 24
cserepufferezés, 51

dedikált
    hálózat kiszolgálás, 111
    rács, 72
digitális, 12

driver, 31

EC mód, 23
EFS, 62
elszámolási rendszer, 19
erőforrás, 15, 39, 82-85
Ext2SF, 60
Ext3, 60
ExtFS, 60

éheztetés, 87

FAT12, 61
FAT16, 61
FAT32, 62
fájl, 61
fájlrendszer, 61
felhő, 118
folder, 59
folyamat, 77
folyamatokból álló rendszer, 77
FORK, 77
FTPFS, 64
fürtözés, 69

garbage collection, 45
gépi utasítás, 12
GPU, 67
    grafikus processzor, 66
grafikus processzor, 66
GRID, 71

gyorsítótár, 52

hangolás, 41, 52, 96, 101
hálózat kiszolgálás
    dedikált, 111
    nem dedikált, 111
HFS, 56
hibernálás, 37

hideg indítás, 37
holtpont, 82, 83
holtponti terület, 85
host, 111
HPFS, 62
Hyper-V, 113

időosztás, 40
időszeletelés, 40
indítás
  hideg, 37
  meleg, 36
installálás, 113
INT utasítás, 22
interpreter, 103
IP regiszter, 24
IPL, 33
ISO 9660, 62

JAVA, 104
JAVA motor, 104
JCL, 95
JES2, 57, 70
JES3, 57, 70
JFS, 62
JFS2, 62
job, 55, 95
JOIN, 77
Joliet, 62
JVM, JAVA virtuális gép, 104

kapcsolatszerkesztő, 101
képfájl, 117
kliens-szerver, 80
kommunikáció, 18, 95
koprocesszor, 22
könyvtár, 59
kötegelt, 55

lapozás, 46
  újabb esély, 47
  előretekintő, 46
  FIFO, 47
  igény szerinti, 46
  LFU, 47
  LRU, 47
  NFU, 47
linkage editor, 101
LPAR, 109

magasszintű nyelv, 100
mappa, 55
massive parallel processor, 68
Master Boot Record, 35
MCM, 68
meghajtó program, 37
megszakítás, 16–31, 49, 67, 77, 108, 114
megszakítás rendszer, 16
megvalósítható állapot, 82
megvalósíthatatlan állapot, 82, 84
meleg indítás, 36
Memory Access
  Cache Only, 69
  Non-Uniform, 69
  Uniform, 69
memória, 12
memóriagazdálkodás, 43
memóriavédelem, 14
mező, 50
mnemonikus kód, 100
moduláris, 20
monolitikus, 20
monopolizálás, 39
mp3, 61
mpeg2, 61
mpeg4, 61
MPI, 78
MPP, 68
multi chip modul, 68
multiprogramozás, 14, 15
MVS, 49, 70

napló, 18
naplózás, 18
nem dedikált
  hálózat kiszolgálás, 111
  rács, 72
Neumann-elv, 2
Neumann-elvű számítógép, 12
NFS, 63
Non-Uniform Memory Access, 69
NTFS, 62

NUMA, 69

online, 56
operációs rendszer, 11
OS/2, 113
overdrive processzor, 66

P2P, 96
partíció
  memória partíció, 43
  merevlemez partíció, 43
párbeszédes, 55
PowerShell, 95
precedenciagráf, 79
prioritás, 39
  belső, 39
  külső, 39
privilegizált
  állapot, 16
  utasítás, 16
Processor Complex, 11
processzor
  CPU, 67
  GPU, 67
  grafikus processzor, 67
  koprocesszor, 67
  overdrive processzor, 67
  többmagos, 68
  társprocesszor, 66, 67
program, 12
programgenerátor, 103
PSW, 23
puffer, 50
pufferezés, 50
pufferezés
  cserepufferezés, 50
PVM, 78

RAID, 60, 63, 116
rács, 71
rács
  dedikált, 72
  nem dedikált, 72
reentrant, 102
regiszter
  állapotregiszter, 24
  CS, 24
  IP, 24
ReiserFS, 62
rekord, 50
Remote-boot, 36
rendező program, 105
ReSF, 63
reusable, 102
REXX, 95
rezidens, 17, 43
ROM, 12
ROM BIOS, 35

sajátidő, 83
sejtautomata, 67
SETI, 78
SFS, 62
shell script, 95
single system image, 69
SMP, 69
spool, 56
SSD, 64
stand-alone, 34, 113
start-stop idő, 83
starter, 113
SVC utasítás, 22

számítóháló, 71
szerkesztés, 100
szerkesztő program, 100
szerkesztőprogram, 100
szerver, 111
szimmetrikus multiprocesszoros, 69
szuper operációs rendszer, 107

TASK, 77
tárolt programozás, 12
társprocesszor, 66
többmagos processzor, 66
tömörítés
  veszteséges, 61
  veszteségmentes, 60
túlcímzés, 22
túlcsordulás, 22
tranziens, 43

UDF, 62
UFS, 63
UMA, 69
Uniform Memory Access, 69
user-mode-linux, 110

veszteséges tömörítés, 61
veszteségmentes tömörítés, 60
VFS, 62
VIO, 49
Virtual PC, 110
virtuális
    címzés, 48
virtuális
    CD/DVD, 117
    felhasználó, 118
    gép, 107, 110, 117
    hálózat, 117
    képernyő, 117
    mágmeslemez, 116
    memória, 45, 115
    nyomtató, 117
    periféria, 115
    processzor, 115
VLAN, 117
VM, 109
VMware, 109
VSAM, 64

wine, 110
WinFS, 63

# *IRODALOMJEGYZÉK*


[1] Benkő Tiborné (1992): *Könnyű a WINDOWS-t programozni ? I-II.* Budapest: Computer Books Kiadói Kft.
[2] Bakos Tamás–Zsadányi Pál (1993): *Operációs rendszerek I-II.* Budapest: Számalk.
[3] Bartók Nagy János–Laufer Judit (1994): *UNIX felhasználói ismeretek.* Budapest: Openinfo.
[4] Benyó Balázs és mások (2000): Operációs rendszerek mérnöki megközelítésben. Budapest: Panem.
[5] Nigel Bryant (1989): *Szakértői rendszer felépítése*. Budapest: Novotrade Rt.
[6] Clarence B. Germain (1988): *IBM PC XT/AT programozói kézikönyv.* Budapest: Novotrade.
[7] Demetrovics János és mások (1989): *A számítástudomány matematikai alapjai.* Budapest: Tankönyvkiadó.
[8] Dobó Csaba–Pászor János (1983): *Virtuális rendszerek és az OS/VS1 kezelése.* Budapest: Számalk.
[9] John J. Donovan (1977): *Rendszerprogramozás*. Budapest: Műszaki.
[10] Fekete István–Gregorics Tibor–Nagy Sára (1990): *Bevezetés a mesterséges intelligenciába.* Budapest: LSI.
[11] Fodor Zsuzsa (1992): *VMS operációs rendszer*. Pécs: CARBOCOMP.
[12] Gerl Zsolt (1985): *Operációs rendszerek időosztásos üzemmódjai I-II*. Budapest: Számalk.
[13] Ian Foster–Carl Kesselman (1999): The Grid: *Blueprint for a New Computing Infrastructure.* San Francisco: Morgan Kaufmann Publishers, Inc.
[14] IBM: *MVS dokumentációk.*
[15] IBM: *OS/2 dokumentációk.*
[16] IBM: *VM dokumentációk.*
[17] Kovács Győző (1997): *Magyar feltalálók, találmányok: Neumann János.* Budapest: Műszaki.
[18] Mary S. Gorman–S.Todd. Stubbs (2003): *Operációs rendszerek.* Budapest: Panem.
[19] MICROSOFT: *DOS dokumentációk.*
[20] MICROSOFT: *Windows dokumentációk.*
[21] Peter Norton (1990): *DOS kalauz*. Budapest: Novotrade.
[22] Peter Norton (1990): *Az IBM PC programozása*. Budapest: Műszaki.
[23] Sára Attila (1978): Számítógép hardver. *TTK egységes jegyzet.* Budapest: Tankönyvkiadó.
[24] William R. Stanek (2005): Microsoft Windows parancssor. *A rendszergazda zsebkönyve.* Budapest: SZAK.
[25] Szenes Katalin–Úry László (1990): *IBM PC DOS I-II-III.* Budapest: LSI.
[26] Tannenbaum Andrew S.–Woodhull Albert S. (1999): *Operációs rendszerek*. Budapest: Panem–Prentice Hall.
[27] Varga László (1978): *Rendszerprogramok elmélete és gyakorlata*. Budapest: Akadémiai.
[28] Vid Ödön (1980): OS programozói segédlet. Budapest: Számalk.
[29] Vid Ödön–Márkon Gábor (1982): *OS programozói segédlet VS kiegészítésekkel*. Budapest: Számalk.
[30] Yoshiaki Shirai–Jun-ichi Tsujii (1987): *Mesterséges intelligencia.* Budapest: Novotrade Rt.