SZAKDOLGOZAT

Sziklási Béla

Debrecen
2009

Debreceni Egyetem
Informatika Kar

Mikrokerneles és monolitikus felépítésű operációs rendszerek összehasonlítása

Témavezető:
Dr. Fazekas Gábor
egyetemi docens

Készítette:
Sziklási Béla
Programtervező Informatikus

Debrecen
2009

Tartalomjegyzék

1. Bevezetés **4.**
1. a, Egy kis történelem 4.
1. b, Célkitűzés 12.
2. Az elmélet **15.**
2. a, Alapok 15.
2. b, A monolitikus kernel 21.
2. c, A mikrokernel 26.
2. d, Alapvető különbségek és hatásaik 30.
2. e, Más modellek 45.
3. Összefoglalás **47.**
4. Irodalomjegyzék **50.**
5. Függelék **51.**
6. Köszönetnyilvánítás **57.**

1. Bevezetés

1. a, Egy kis történelem

A közvélekedés szerint amióta számítógépek vannak, léteznek operációs rendszerek is. – Ez persze nem egészen igaz, hiszen például a Harvard Mark I óriásgépen (ami az általános felfogás szerint a legelső elektronikus – vagy inkább még elektromechanikus – digitális számítógép) még nem beszélhetünk ilyesmiről. Azonban a számítógépek második generációjánál (kb. 1956-tól) már megjelentek az első olyan programok melyeknek a feladata a többi, a felhasználók (akkoriban szinte még kivétel nélkül programozók) számára „értékes" program futásának felügyelete – ők voltak a mai operációs rendszerek ősei. Azóta hellyel-közzel igaz az állítás miszerint minden számítógéphez tartozik egy „OS", és számítógép alatt manapság már nem csak a fél íróasztalt elfoglaló PC-t, vagy egy esetleg még ennél is nagyobb szerver-, ipari- vagy tudományos célokra készült célgépeket kell értenünk, hanem az életünk egyre szervesebb részét képező hordozható eszközöket is, a laptopoktól kezdve egészen a mobiltelefonokig - és persze ki tudja mit hoz a jövő, hiszen ma már a hűtőszekrényünk internetezik, a walkman „leszármazottai" pedig általános célú multimédiás és kommunikációs központok. (Természetesen az operációs rendszer nem mindig egy „kézzel fogható", jól elkülöníthető dolog, mivel sokszor nagyon szorosan kapcsolódik a hardverhez, amivel árukapcsoltan adják. Számos architektúrának megvolt/megvan a saját, egyedi, sehol máshol nem alkalmazott és lecserélhetetlen - esetleg közvetlenül a gép ROM-jába égetett – operációs rendszere. Ilyenek voltak például bizonyos régebbi gyártmányú mobiltelefonok, vagy a Magyarországon is igen népszerű Commodore számítógépek.)

Persze a hardhez és a felhasználói programokhoz hasonlóan az operációs rendszerek is óriási fejlődésen mentek keresztül a kezdetek óta eltelt nem is oly hosszú idő alatt. Míg az első proto-rendszerek funkcionalitása kimerült abban, hogy egy szalagról sorban a memóriába töltötte és elindította a programokat, manapság egy operációs rendszer magas szintű esztétikai élményt nyújt, és szinte gondolkodik is helyettünk. (Legalábbis ha hinni lehet a marketing szlogeneknek.) Ahogyan gazdaságossági és kényelmi szempontok miatt realizálódott az igény hogy egy számítógépen egyszerre több program is futhasson, úgy váltak az ennek biztosítását

feladatul kapó operációs rendszerek is egyre bonyolultabbakká. („Egyszerre” alatt valódi párhuzamos futást persze csak egy többprocesszoros rendszeren érthetünk). Ráadásul ezek programok (az operációs rendszer szemszögéből: folyamatok) általában nem egyetlen felhasználóhoz tartoztak, ami újabb feladatokat rótt az operációs rendszerre: egyrészt meg kellett védeni a felhasználók adatait egymástól, másrészt egy másfajta védelemről is gondoskodnia kellett: régebben ha egy hibásan megírt program összeomlott, lefagyott, azaz folytatásra alkalmatlan állapotba került akkor a helyzet megoldható volt a számítógép újraindításával és egy új program betöltésével, kára csupán a hibás programot író programozónak származott az egészből. (Ez talán nem egészen igaz, de legalább más munkája, adatai nem vesztek el.) Multiprogramozott környezetben azonban már nem volt semmiképpen megengedhető, hogy egyetlen program hibája olyan állapotba hozza a rendszert, hogy a többi futása se tudjon folytatódni. Ezzel az igénnyel párhuzamosan jelent meg egy másik is a programozók részéről (akik ekkor már nem kizárólagos használói voltak a gépeknek). A számítógépek harmadik generációjának idején a hardver már jóval bonyolultabb és főleg változatosabb volt, mint a kezdeti időkben, amikor minden programba bele voltak kódolva a hardver közvetlen eléréséhez és működtetéséhez szükséges utasítások. Egyre több és egyre összetettebb programok készültek a növekvő felhasználótábor (akkoriban még főleg tudományos projectek és az üzleti szektor) részére, és a programok készítői szerették volna, ha a hardver elérésének „finomságaival” nem nekik kell foglalkozni, hanem azt egyszerűbb, egységes és szabványos módon tehetik meg azt az operációs rendszer szolgáltatásainak (úgynevezett rendszerhívásoknak) a segítségével. Ennek kettős előnye volt: egyrészt egyszerűbbé vált a programok megírása, másrészt könnyebb is volt őket átportolni egyik gépről a másikra. (Pontosabban ideális esetben nem is kellett portolni csak az operációs rendszert. Ebben az időben már léteztek ún. számítógépcsaládok, melyek minden újabb tagja lefelé kompatibilis volt a régiekkel, de ez alapjában véve csak a processzor utasításkészletének azonosságát jelentette, a perifériákét nem.) Mindezeket a célokat tehát úgy lehetett elérni, hogy a felhasználói programoktól meg lett tagadva a perifériákat közvetlenül író/olvasó műveletek végrehajtásának a joga, ezeket az eszközöket egy az OS által biztosított, jól definiált interfészen keresztül érhették el. (Az angol rövidítés API: Application Programming Interface). A mechanizmus a következő: a futó program valamilyen speciális utasítás (például egy szoftveres megszakítás-hívás) segítségével elindítja az operációs

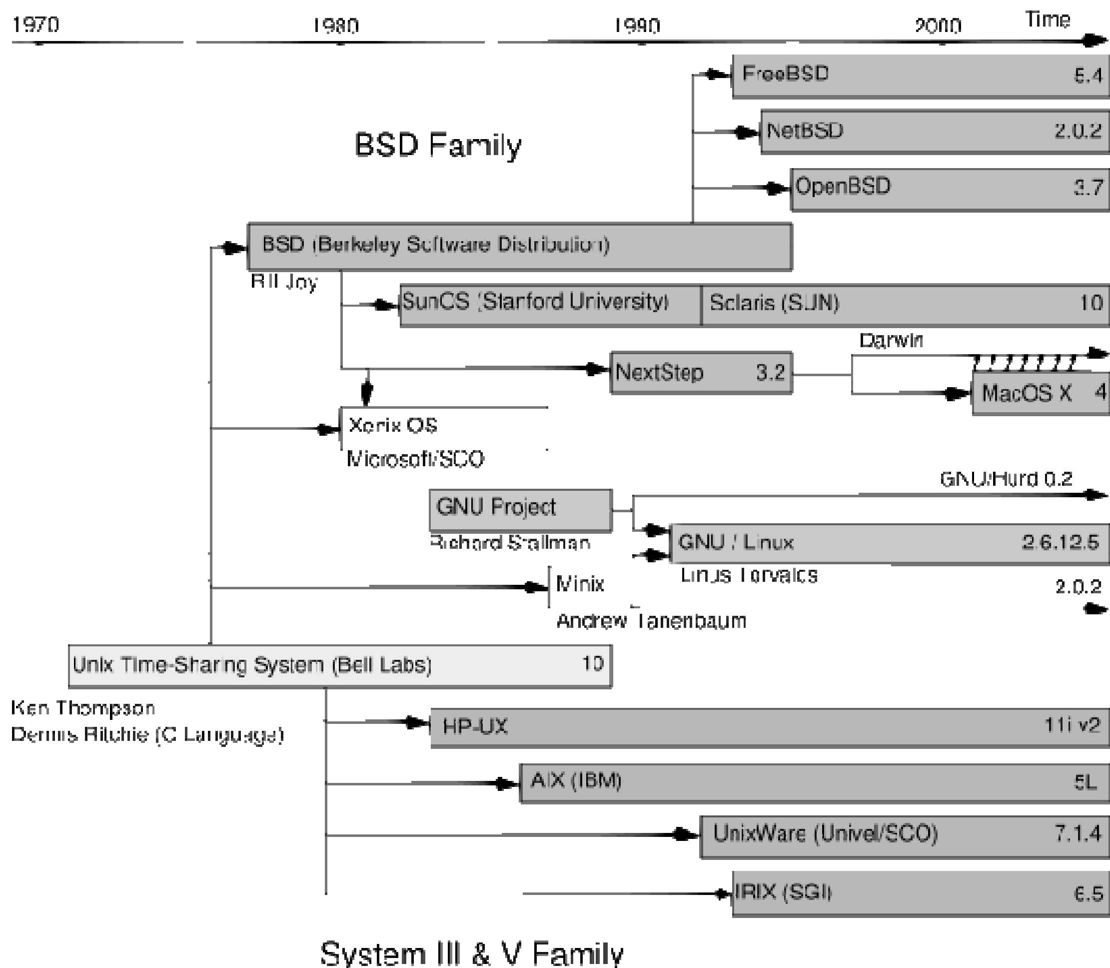
rendszer megfelelő szolgáltatását, átadva neki (processzor regiszterben, veremben stb.) a végrehajtáshoz szükséges paramétereket, majd az operációs rendszer a megfelelő ellenőrzések után végrehajtja azt, és csak az „eredményeket” (hibakód, memóriacím stb.) adja vissza a hívó programnak. (Annak kikényszerítése, hogy a futó felhasználói program ne hajthasson végre „privilegizált”, azaz a saját memóriaterületén kívül eső címeket vagy a hardvert közvetlenül elérő utasításokat egy, a harmadik generációs gépek processzoraiban megjelenő újítással – pl. „védett mód” az x86-os processzorcsaládban - érhető el: a processzor több különböző privilégium szinten képes futni, amelyből csak a legfelsőben hajlandó direkt I/O utasításokat végrehajtani, arra azonban csak azután vált át, miután a futó felhasználói program a megfelelő speciális utasítással meghívta az operációs rendszer egyik rutinját. – Ez a koncepció képezi tulajdonképpen az alapját ennek a dolgozatnak is, lévén a különböző operációs rendszerek különböző módon kívánják ezt az „adottságot” kihasználni. Amelyik architektúra nem rendelkezik ilyen vagy ehhez hasonló képességgel, ott biztonságos operációs rendszer csak egy jóval körülményesebb módon írható: ott az OS végigmegy a felhasználó program végrehajtható kódján utasításról utasításra, mindegyiket megvizsgálja, értelmezi, és aztán (ha biztonságosnak ítéli) végrehajtatja a processzorral. A program kódja, mely igazából nem fut soha közvetlenül a processzoron, nem is feltétlenül gépi kód, lehet valamilyen byte-kód is. Ezzel a módszerrel valóban bomba-biztos rendszer írható, a rendkívüli teljesítménybeli hátrányok miatt azonban nem használatos olyan architektúrán, amin egyébként támogatott a védett mód - még a mai számítási teljesítmények mellett sem. Persze nem lehet kizárni azt sem, hogy egy ilyen interpreter-szerű megoldás lesz a jövő útja, látva például a hasonló lábakon álló Java technológia népszerűségét. Manapság már rengeteg helyen használnak „virtuális rendszereket”, ahol egy emulált számítógépen fut az egész operációs rendszer is, nem csak a felhasználói programok. Tekintve, hogy mostanában egyre általánosabb trend a nagyközönségnek szánt programoknál a „kényelem előrébbvaló a teljesítménynél” elv, nem zárható ki hogy pár év múlva ennek a dolgozatnak a témája is tárgytalan lesz ekképpen. Nem teljesen persze, hiszen egy ilyen rendszernél is felmerül a kérdés, hogy az értelmező kódjának mekkora és mely részét érdemes védett módban futtatni, illetve az operációs rendszer mely szolgáltatásai kerülhetnek a feldolgozott felhasználói kódba. Ettől függetlenül azért egy ilyen rendszer felépítése és működése alapvetően eltérne az itt tárgyalandó jelenkoriakétól, de széleskörű elterjedését a közeljövőben személy szerint nem tartom valószínűnek. Mindezidáig

a tapasztalat azt mutatta, hogy a hardver sosem tudott gyorsabban fejlődni, mint a felhasználói igények, és ez a növekedés napjainkban inkább lassulni látszik, mint gyorsulni, még akkor is, ha a többprocesszoros, illetve többmagos asztali rendszerek elterjedése változtathat ezen a trenden – legalábbis a munkaállomások piacán. A múltban több jóslat született arra vonatkozólag, hogy a teljesítmény helyét rövid időn belül végleg átveszi a biztonság az operációs rendszerek fejlesztésénél használt prioritás-lista első helyén, de ez máig nem történt meg. Eleddig ugyanis a vezető fejlesztők közül senki nem merte bevállalni, hogy egy az előzőeknél esetleg nagyságrendekkel lassabb rendszert dobjon piacra a megnövelt stabilitás áraként. Nem kereskedelmi alapú fejlesztéseknél persze előfordultak ilyen próbálkozások is, de az ilyen projecteknél általában egyaránt hiányzik a kezdő felhasználói bázis és egy teljes funkcionalitású operációs rendszer kialakításához szükséges fejlesztői/tesztelői erőforrások, így sosem lesz belőlük számottevő piaci részesedéssel bíró platform.)

Egy folyamat nem csak úgy tudja a többi működését végleg megakasztani, ha szabálytalanul használja az erőforrásokat, hanem úgy is ha „örökre" lefoglalja a processzort, például mert végtelen ciklusba került. (Azt a modellt, ahol az operációs rendszer nem veheti el a CPU-használat jogát egy folyamattól, hanem meg kell várnia, míg az önként lemond róla „non-preemptive multitasking-nak" nevezi az angol szakirodalom). Az első multiprogramozott rendszerekben ez még nem merült fel annyira komoly problémaként, mivel amúgy is csak viszonylag kevés váltás történ az aktív folyamatok között: az egyetlen kitűzött cél a processzor minél jobb kihasználása volt, ezért egy folyamat csak akkor mondott le róla mikor I/O-ra kellett várnia. Később azonban megjelent a multiprogramozás egy magasabb foka: az időosztásos rendszerek. (Bizonyos definíciók ezt már nem is a multiprogramozásként fajtájaként értékelik, hanem teljesen külön fogalomként.) Egy időosztásos rendszert egyszerre általában több felhasználó használ, akik mind a saját külön termináljukon keresztül kapcsolódnak egy viszonylag nagy teljesítményű számítógéphez, és mind elvárják, hogy a - lehetőségekhez képest – úgy tűnjön mintha a rendszer csak az ő programjukkal/programjaikkal foglalkozna. Ehhez már mindenképpen szükség van a fenti probléma megoldására, ami meg is történt különböző ütemezési stratégiák, és később a „preemptive multitasking" bevezetésével. (Ezek itt most nem lesznek részletezve.)

1964-ben három akkoriban igen jelentős számítástechnikai cég (a Massachusetts Institute of Technology, a General Electrics és a Bell Labs) belekezdett egy közös projectbe,

melynek a célja egy ilyen „egy központi gép sok terminállal” időosztásos rendszer megépítése volt, amihez saját operációs rendszert is terveztek és készítettek: ez volt a MULTICS. Annak ellenére, hogy az eredeti project megbukott, és maga a MULTICS sem vált széles körben elterjedtté, mégis hatalmas hatással volt a számítástechnika, azon belül is az operációs rendszerek fejlődésére. A MULTICS ötleteiből kiindulva alkotta meg később a Bell Labs-nál Ken Thompson, Dennis Ritchie és még jó pár másik programozó azt a szoftvert, amit aztán UNIX-nak neveztek el. A UNIX szolgált mintául a legtöbb mai - asztali munkaállomásokon és webszervereken futó - operációs rendszer számára, és szolgáltatásaival valamint tulajdonságaival azok számára is egy standardot szolgáltatott, melyek nem követték struktúrában és működésben.



A „UNIX – típusú” rendszerek családfája

A UNIX az eleinte nyílt, C nyelvű forráskódjának köszönhetően könnyen portolható volt különböző rendszerekre, és széleskörű szolgáltatásainak köszönhetően hamar elterjedtté és népszerűvé vált számos hardver-platformon, köztük a 80-as évek közepére a személyi számítógépek rohamosan növekvő gazdasági jelentőséggel bíró piacán lassacskán domináns helyzetbe kerülő Intel x86 architektúrán is. (A processzorcsalád az IBM PC megjelenésével indult el hódító útjára, és mára elsöprő fölénnyel uralja a nagyobb hordozható gépek, az asztali munkaállomások és a kisebb szerverek piacát. Mivel ezek azok a területek, amik gyakorlati jelentőségük miatt elsősorban vizsgálódásom tárgyát képezik, ezért ha külön másként nem jelzem innentől az x86 architektúrán futó operációs rendszerekre vonatkozik majd minden megjegyzésem.)

Az IBM PC architektúrán a UNIX variánsokat csak a munkaállomások piacán sikerült megszorítania egy jelentős vetélytársnak, az MS-DOS-nak, illetve annak más neveken megjelenő, de amúgy nagyrészt azonos változatainak. A DOS közel sem nyújtott annyi szolgáltatást mint a UNIX, de egyszerűsége miatt mégis igen népszerűvé vált a számítógéphez kevésbé értő otthoni felhasználók között. Lényegében ma is e két rendszer közvetlen vagy közvetett leszármazottai uralják az x86 architektúrát, és ezáltal lényegében az általános célú számítógépek (pontosabban azok operációs rendszereinek) piacát.

A Microsoft által fejlesztett rendszerek esetén a családfa viszonylag egyszerű, lévén a szoftverrel kapcsolatos jogok is mind a cég kezében vannak, így maga a fejlesztés is központilag (üzleti alapon) meghatározott módon történt. Az operációs rendszer újabb és újabb verziói kényelmes időközönként követték egymást, mindig továbbfejlesztve és fokozatosan kiszorítva az előzőt. (Eltekintve egyetlen viszonylag rövid időszaktól mikor a korai NT és a Windows 9x termékvonal egyszerre volt a piacon, kvázi egymással „konkurálva”.) Az MS-DOS-t több mint 10 évnyi fejlesztés után 1994-ben nyugdíjazták, de lényegében még 2000-ig tovább élt, összeolvasztva a megújított felületű Windows keretrendszerrel, mint grafikus felhasználói felülettel, és egy kicsit átdolgozott kernellel. (Hogy jobban kihasználhassa az akkor már 32 bites Intel processzorok képességeit). A Microsoft annak alapvető korlátai miatt 2000 után végleg eldobta a DOS kernelt, és a már jóval korábban kifejlesztett, modernebb elveken alapuló NT kernelre építve adta ki a régi felhasználói- és rendszerhívás-interfészét megtartó Windows termékcsalád újabb verzióit, az XP-t (2001-ben,) a Vista-t (2006-ban,) és a Windows 7-et (2009?) Ez idő alatt a Microsoft

kezdeti kedvező pozíciójából monopólium-közeli helyzetet harcolt ki magának a személyi számítógépek piacán – többek között jó marketingstratégiájával és a felhasználó-barátiságot minden egyéb szempont elé helyező fejlesztési elveinek következetes betartásával.

A fentiekből részben következik, hogy UNIX leszármazottak sorsa egy kissé hányatottabb volt. Ma már nincs aktívan fejlesztett operációs rendszer melynek neve „UNIX" lenne, van viszont jó pár (részben kommerciális, részben nyílt forráskódú) rendszer mely továbbviszi az örökséget. A Windows után egyértelműen a legnagyobb piaci részesedéssel rendelkeznek, de a különböző változatok más és más felhasználási területen bírnak kiemelkedő jelentőséggel. Talán elsőként említendő a Linux, amely amellett hogy említésre méltó részesedéssel bír a munkaállomások piacán a szerveroldalon megkerülhetetlen ha UNIX alapú megoldásokról van szó. A Linux egy GPL licenc alá eső kezdeményezés, mely valójában nem használ fel egy sort sem az eredeti UNIX forráskódból (jogi okok miatt), de pontosan betartja a POSIX IEEE szabványt, amely megadja, hogy milyen rendszerhívásokkal kell egy UNIX kompatibilis operációs rendszernek rendelkeznie. (Gyakran vita tárgya hogy a Linux kernel önmagában, a GNU project keretében fejlesztett kiegészítő programok nélkül nevezhető-e operációs rendszernek vagy csupán egy rendszermag, de ez itt most irreleváns.) A szerverek piacán szintén fontos szerepe van a Sun Microsystems által fejlesztett Solaris-nak, (korábban SunOS,) melyben számos jelentős technikai újítás először jelent meg, és melynek forráskódját nem régen szintén megnyitották a nyilvánosság és a fejlesztő-közösség előtt. Mindenképpen szót érdemel a UnixWare is, az SCO Group által forgalmazott Unix változat mint az eredeti UNIX rendszer talán legközvetlenebb leszármazottja. Szintén jelentősek (főleg szervereken) a UNIX oldalhajtásának, a BSD-nek a nyílt forráskódú leszármazottai, mint az OpenBSD, a FreeBSD, vagy a NetBSD. Megemlíthető még a Prof. A Tanenbaum által írt Minix-et, melynek gyakorlati jelentősége ugyan csekély, de oktatási szerepe annál nagyobb, és a Windows után legelterjedtebb „asztali" operációs rendszer, az Apple tulajdonában lévő MacOS X, mely ugyan POSIX -szabvány szerinti, de felépítésében talán a legtávolabb esik a hagyományos UNIX által felállított modelltől.

És akkor álljon itt még egy kis történeti érdekesség, ami igen szorosan kötődik dolgozatom témájához: a '90-es évek elején, abban az időben, amikor a hardver fejlődési üteme tetőpontjához közeledett, komoly vita tárgya volt a fenti rendszerek fejlesztői körében hogy az OS központi magjának, a kernelnek, melyik strukturális modellt kellene követnie.

Elméleti szempontból a mikrokernel jóval több előnyt kínált, de praktikus okok miatt sokan ragaszkodtak a monolitikus felépítéshez is. (Ezekről részletesen majd később). Ennek a vitának volt egy tipikus példája két fenti rendszer, a Linux és a Minix alkotójának szócsatája az Internet nagyközönsége előtt, melyben Tanenbaum professzor a Linuxot már készülte idején „elavultnak” nevezte, két okból: egyrészt monolitikus felépítése miatt, ami „nem a jövő útja”, másrészt mert túl szorosan kötődik (?) az x86 architektúrához (és annak is 32 bites változatához) mintsem hogy portolható legyen, holott az (jóslata szerint) hamarosan ki lesz szorítva más processzortípusok által. Mint az utólag kiderült a professzor tévedett mindkét érvében: nemcsak az azóta sokkal népszerűbbé váló Linux használ még ma is monolitikus felépítésű kernelt, hanem a legtöbb fentebb sorolt Unix variáns is, sőt bizonyos szempontból a Windows-é is annak tekinthető. (Nem levonva ebből semmi további következtetést!) Másrészről az x86 architektúra (illetve lassan már annak 64 bites variánsa) most is legalább annyira egyeduralkodó, mint akkoriban volt, ráadásul a Linux ennek ellenére azóta gond nélkül át lett portolva egy sor másik platformra is. Ez a fejezet csak némi felvezető szándékozott lenni, nem kapcsolódva szorosan a dolgozat témájához, de megvilágítva néhány megkerülhetetlen alapkoncepciót, és bemutatva a probléma történelmi hátterét és előéletét. Olyan operációs rendszerek lettek benne megemlítve melyek komoly gyakorlati jelentőséggel is bírtak, akár közvetlenül, (elterjedt használatuk miatt,) akár közvetve (későbbi jelentős rendszerekre gyakorolt hatásuk miatt). A dolgozat elemző részében főként ezekre a rendszerekre fogok utalni mint nagy gyakorlati jelentőségű megvalósításaira az boncolgatott modelleknek, néhány kevésbé ismert de elméleti szempontból érdekes operáció rendszerrel együtt.

1. b, Célkitűzés

Ez a dolgozat nem elsősorban egy gyakorlati útmutató akar lenni. Nem kizárólagos célom volt két vagy több konkrét operációs rendszert összehasonlítani abból a célból, hogy eldöntsem melyik használata üdvösebb ilyen vagy olyan környezetben. Két alapvető rendszer-modellt, két tervezési technikát, két filozófiát szerettem volna megismerni és bemutatni. Elemezni fogom a két kerneltípust képességeik, nyújtott szolgáltatásaik, és tulajdonságaik alapján. Megpróbálom felderíteni az operáció rendszerek fejlesztésében jelentkező múltbeli és jelenkori trendeket, és megállapítani ezek lehetséges kihatásait a szoftveripar ezen szegmensére. Az elemzést az elméleti síkon kezdem majd, a két technológia absztrakt modelljével, és később térek majd rá az egyes funkciók gyakorlati megvalósításának a bemutatására konkrét rendszerekben.

A monolitikus modell leginkább a Linux példáján keresztül lesznek bemutatva, mivel az egy könnyen hozzáférhető, nyílt és jól dokumentált forráskódú rendszer, a modern operációs rendszerek minden alapvető funkciójával ellátva, és mivel klasszikus (bár moduláris) monolitikus kernellel rendelkezik.

A mikrokerneles rendszerek esetén nem láttam értelmét egyetlen példa kiragadásának, hiszen ezen a téren nem nagyon beszélhetünk (még?) piacvezetőről a munkaállomások/általános célú szerverek piacán, mint ahogy azt sem lehet megjósolni, hogy a sokszor igen eltérő elveken alapuló rendszerek közül melyik mekkora hatással lesz majd a későbbiekre. Ezek közül a rendszerek közül mindenképpen említésre érdemes azért a Mach3, egyrészt történelmi jelentősége miatt, másrészt mert arra épül a GNU/Hurd operációs rendszer, illetve az XNU, a Mac OS X rendszer magja is, mely így még mindig a legelterjedtebb ilyen OS a személyi számítógépek piacán. (A Mac OS X nem rég lett átportolva az x86 architektúrára is. Pici szépséghiba a dologban hogy az XNU valójában nem igazi mikrokernel, bár abból lett kifejlesztve. Ha olyan mikrokerneles rendszerről szeretnénk szót ejteni, amit elterjedten használnak a gyakorlatban is akkor a mobil operációs rendszerek piacán kell keresgélnünk, valahol a Symbian és a QNX környékén. Ezekre viszont bővebben nem térek ki rá, mivel a mobil rendszerek nem témái ennek a dolgozatnak, lévén teljesen más követelményrendszernek kell megfelelniük. Az ilyen beágyazott rendszereknél általában eleve más motivációk húzódnak meg a mikrokerneles megoldás használata mögött, mint az asztali

és szerver rendszereknél.) Jelentős még ebben a kategóriában az L4-et mint a modern, második generációs mikrokernelek egy tipikus példája, és a Minix, mely oktatási célból készült rendszer lévén szándékolt egyszerűsége és strukturáltsága (és a készítője által hozzáfűzött bőséges kommentár) miatt különösen alkalmas az elméleti modell illusztrálására. Speciális okból szeretnék majd egy gondolatmenet erejéig kitérni még egy mikrokerneles rendszerre, ez pedig a Microsoft berkein belül „Singularity" kódnéven futó project „végterméke". A fenti szempontok alapján a gyakorlati jelentősége kicsiny ugyan, lévén még csak kísérleti/teszt jelleggel használják, illetve fejlesztenek rá. Ennek ellenére úgy vélem mindenképpen megérdemel egy – a lehetőségekhez mérten – alaposabb vizsgálódást a téma, mégpedig két okból: egyrészt úgy tűnik, számos olyan forradalmi újdonságot tartalmaz, amik idővel alapvető hatással lehetnek az „iparágra," másrészt tekintve a fejlesztő cég piaci pozícióját könnyen előfordulhat, hogy a közép-távoli jövőben még komoly gyakorlati szerepben látjuk viszont.

Természetesen hivatkozni fogok a Windows termékcsalád újabb tagjaira is. Egyfelől a Windows NT kernel egy jó példa a két modell közötti átmenet egyik lehetséges megvalósítására, (a másik széles körben elterjedt megvalósítás a Mac OS X-é,) másfelől elterjedtségükből adódó gyakorlati jelentőségük miatt amúgy is elkerülhetetlen az érintésük egy olyan tanulmányban, ami akár csak a látszatát is meg akarja őrizni annak, hogy nem teljesen a „földtől elrugaszkodva", elmélet síkon mozog.

A kép teljessége kedvéért a címben szereplő két modellen és a kettő közötti átmeneti megoldásokon kívül említést teszek majd olyan „extremitásokról" is mint a nano-, és exo-kerneles rendszerek, de részletekben nem térek ki rájuk, mivel pillanatnyilag nem látszik valószínűnek, hogy bármelyik technológiát is komolyabb gyakorlati jelentőségre tenne szert a közeljövőben. Ezek inkább csak tudományos érdekességek.

A téma, amit választottam valószínűleg nem tűnik húsbavágó fontosságúnak praktikus szempontokból. Tulajdonképpen nem is az. Ezzel együtt több oka is van annak, hogy e mellett döntöttem, ezek közül is az első és legfontosabb, hogy személy szerint érdekel. Szerény vélemény szerint egy jó programozó (vagy akár általánosabban: informatikus) a számítógéppel kapcsolatban nem gondolkodhat úgy, hogy megelégszik azzal, hogy tudja hogyan kell használni, és nem érdekli hogyan és miért *működik*. Ennek megértéséhez viszont az út elkerülhetetlen módon (többek között) az operációs rendszer és a hardver működésének

megismerésén keresztül vezet. Ezen belül pedig a címben említett téma egyike a legszebb, elméleti és gyakorlati aspektusokkal egyaránt bőven rendelkező problémáknak.

Más szemszögből nézve pedig igenis van gyakorlati haszna is. Egy átlagos felhasználót nem érdekli, hogy belül hogyan működik az operációs rendszer, és ez így van rendjén, nem is kell, hogy érdekelje, hiszen nincs rá szüksége, hogy tudja. Az OS szempontjából pedig az átlagos programozó is csak egy felhasználó, aki igénybe veszi a szolgáltatásait, de nem lát és nem nyúl beléjük. Ugyanakkor annak viszont, aki rendszerközelibb programokat, vagy egy nagyméretű szoftver rendszerközelibb moduljait készíti igenis hasznos, ha ismeri a belső mechanizmusokat, ahhoz hogy hatékonyabb és megbízhatóbb kódot írhasson. Ráadásul a közelmúlt kezdeményezéseit figyelembe véve elképzelhető hogy a hardver fejlődése, és a mikrokernelek folyamatos fejlesztése és optimalizálása a közeljövőben olyan helyzetet fog előidézni, melyben már az utóbbiak is effektíven megjelennek a használatban a monolitikus rendszerek mellett. Márpedig egy olyan számítógéprendszer üzemeltetőjének vagy karbantartójának ahol az utóbbi (vagy mindkét) típusú operációs rendszerek használatban vannak, szintén hasznos lehet a munkavégzéshez, ha ismeri mindkét fajta rendszer belső felépítését, működését és különbségeiket, valamint erősségeiket és korlátaikat.

2. Az elmélet

2. a, Alapok

Bár már jó párszor használtam, de még nem lett pontosan definiálva, pedig alapvető a *folyamat (process)* fogalma: a folyamat lényegében egy végrehajtás alatt lévő program, pontosabban a program betöltött kódjának, aktuális adatainak valamint egyéb, operációs rendszer specifikus állapotleíró információinak összessége. A folyamathoz kapcsolódik a végrehajtási *szál* fogalma is: egy folyamat több, látszólag (egy processzor esetében, időosztással) vagy fizikailag is (többprocesszoros rendszerben) párhuzamosan futó „feladatra" bomlik, ezek a szálak. A folyamatokkal ellentétben az (egy folyamathoz tartozó) szálak megosztják az erőforrásaikat.

Aztán nem árt tisztázni egy látszólag triviális alapfogalmat, ami bár az általános iskolás számítástechnika könyvekben is szerepel, mégis sokszor gondot okoz a behatárolása, ez pedig az *operációs rendszer*. A látszattal ellentétben ez valóban problémát jelenthet, mivel például a szakirodalom sem mindig ugyanarra hivatkozik alatta. Tanenbaumnál az operációs rendszer a privilegizált processzormódban futó rendszermag, a kernel (ami nála lefordítva csupán kilobájtban mérhető méretű) míg például a Microsoft teljes DVD-n árul operációs rendszert, amin a felhasználó kémprogram-írtótól kezdve filmszerkesztőig mindent megkap amire szüksége lehet a számítógép mindennapos használata során. Nálam a kifejezés az összes olyan kód együttesét fogja jelenteni, aminek segítségével a felhasználó vagy a programozó a gépen az adott operációs rendszerre írt programokat el indítani és futtatni tudja. Vagyis ide tartozik a hardver absztrakcióját adó rétegen kívül az összes funkció, ami az operációs rendszertől rendszerint elvárt alapvető szolgáltatásokat nyújtja (fájl-rendszer elérés, hálózat-kezelés stb. – részletesen lásd lejjebb,) akár részei azok a szorosan vett kernelnek, akár felhasználói jogosultságokkal bíró külön folyamatként futnak; és én ide veszem azt az felületet (*interface-t)* is, amin keresztül a felhasználó a rendszerrel kommunikálhat, legyen az egy GUI vagy egy egyszerű szöveges parancsértelmező. (Ennek megfelelően nálam a szorosan vett definíció szerinti Linux nem egy operációs rendszer, hanem csak egy kernel: az

operációs rendszerek a kernelt sok egyébbel együtt tartalmazó disztribúciók – az már más kérdés hogy a köznyelv ezeket összefoglalóan szintén Linuxnak nevezi.)

Ha nagyon tömören akarjuk megfogalmazni, hogy mi is egy OS feladata akkor azt mondhatnánk: kapcsolatot tart a felhasználóval, valamint kezeli az erőforrásokat és a futó programokat. (Ez az „általános iskolás” definíció.) De nézzük meg tételesen mi mindent is csinál egy tipikus OS:

1. Folyamatkezelés: minden modern operációs rendszer támogatja a lehetőséget, hogy a felhasználó „egyszerre” több programot futtathasson. Ez jelentős feladatot ró az operációs rendszerre, lévén neki kell valami stratégia alapján meghatároznia, hogy mikor melyik folyamat kapja meg a jogot a futásra a CPU-n, és mindezt úgy kell megszerveznie, hogy egy programnak se kelljen „örökké” (túl sokáig) várnia, ugyanakkor a felhasználó számára a lehető legkényelmesebb legyen az elosztás. A processzoridő mellet persze jó pár más, párhuzamosan nem használható erőforrást (pl. perifériák) is meg kell osztania a rendszernek. (Többprocesszoros rendszereken a helyzet nem egyszerűbb, hanem még bonyolultabb: azt is el kell dönteni, hogy ki *melyik* processzoron futhat.) Nevezhetnénk ezt akár a legfontosabb feladatnak is, ha nem lenn még egy sor másik, amik nélkül szintén működésképtelen volna a rendszer.
2. Memóriakezelés: ha lehet még a fentinél is összetettebb feladat. (Főleg multitasking rendszereknél az, ami viszont ma már ugyebár alapkövetelmény.) Az operációs rendszernek nem csak azt kell megoldania hogy a folyamatok ne „éheztethessék halálra” egymást, hanem azt is hogy ne olvashassák, és főleg ne írhassák felül egymás tárban lévő adatait (ne adj’ Isten kódját.) Ezt általában a virtuális címterek alkalmazásával érik el. (Az MMU segítségével.) Minden folyamat úgy látja mintha a számára kiutalt RAM lenne a teljes fizikai memória, melynek első bájtja a 0 címen érhető el. Az OS laptáblákat tart fenn, melyek alapján meghatározható a virtuális címből a fizikai. Ennek megfelelően, ha egy régi vagy egy új folyamatnak több tárra van szüksége, akkor azt az operációs rendszertől kérheti a megfelelő rendszerhívással, de ez mindjárt átvezet egy másik problémához: mivel egy multitasking rendszert úgy illik megtervezni, hogy elméletileg akárhány program futhasson egy időben, (kivéve a Windows bizonyos Oroszországban forgalmazott verzióit) így gyakran előfordul, hogy az összes aktív folyamat által igényelt memória mennyisége meghaladja a

rendszerben rendelkezésre állót. Ezt a modern operációs rendszerek az ún. virtuális memória szolgáltatással oldják meg: a RAM-ban lévő adatokból azt, amire valószínűsíthetőleg a legkevésbé lesz szükség kiírja valamilyen lassabb elérésű háttértárolóra (jellemzően merevlemez,) és annak a helyére tölti az újonnan igényelt adatokat. Eközben persze nyilván tartja, hogy mely programhoz tartoztak a kiírt adatok, és ha annak mégis szüksége lenne rá akkor visszatölti valami más helyére, amit az előzőekhez hasonlóan „kilapoz." (A virtuális memória kezelése persze egy ennél jóval összetettebb dolog, kifinomult kiválasztási algoritmusokkal és bonyolult adminisztrációval, de ennek működésébe a folyamatütemezéshez hasonlóan itt most nem megyek bele, mivel a téma szempontjából nincs igazi jelentősége, csupán annak hogy ilyen szolgáltatások léteznek, és implementálni kell őket!)

3. Fájlrendszerek kezelése: amellett hogy az operációs rendszernek gondoskodik a háttértárnak (leggyakrabban mágneses vagy optikai lemez,) mint hardvernek az eléréséről, a felhasználói programok megírásának megkönnyítése végett és biztonsági okokból hozzá tartozik az azon lévő adatok strukturált kezelése és elérése is. A programok az összetartozó adataikat fájlokba, azokat pedig könyvtárakba szervezve szeretik látni, viszont annak nyilvántartása hogy egy adott fájl fizikailag a lemez mely részén vagy részein helyezkedik el már az operációs rendszer feladata. Ide tartozik részben a fizikai lemezhibák miatti esetleges adatveszteségek minimalizálása, megelőzése, lehetőség szerinti kiküszöbölése is.
4. Hálózatkezelés: a számítógép a legkülönbözőbb módszerekkel össze lehet (és jellemzően van) kötve más számítógépekkel. Ennek előnye akkor jelentkezik, ha a másik gépen futó programokkal tudnak kommunikálni a mi gépünkön futó programok. Itt igazából megint nem a hálózati hardver elérésére kell elsősorban gondolni, hanem arra hogy az OS formázza meg az elküldendő, és értelmezi, „dekódolja" a beérkező adatokat. (Hálózati protokollok.)
5. Eszközelérés: Az operációs rendszernek tudnia kell elérni és kezelni a gépben lévő összes hardvert, hiszen a felhasználói programok is csak az ő szolgáltatásain keresztül tudják használni azokat. Olyan eszközökkel is meg kell tudnia birkózni, melyeket „nem ismer", azaz melyeknek elérési módja nincs az operációs rendszer eredeti kódjában definiálva. Ennek megoldására vezették be az illesztőprogram (angolul

„*device driver*”) fogalmát: egy olyan eljárás(gyűjtemény), melynek meghívásával az operációs rendszer feladatkörébe tartozó, az adott hardverre vonatkozó szolgáltatásokat lehet elérni. Lehet, hogy az eredeti rendszerrel volt szállítva, de származhat külső forrásból (pl. az eszköz gyártójától) is. Nagy dilemma hogy szabad-e privilegizált módban futnia: hiszen egyrészt szüksége lehet a hardver közvetlen elérésére, másrészt viszont a fent említett változó eredet miatt nem biztos, hogy meg lehet benne bízni. (Az operációs rendszer azon rétegét mely a hardver elérését biztosítja a többi program számára egy szabványos interfészen keresztül „*Hardware Abstraction Layer-nek*” (HAL) nevezzük).

6. Biztonság: egy általános, átfogó fogalom, ami sok, gyakran alig összefüggő dolgot takar, amik általában nem önmagukban, hanem a fentiekhez kapcsolódva jelentkeznek. Egyrészt a folyamatokat meg kell védeni egymástól: nem szabad hogy egymás működését akadályozzák, megakasszák, vagy egymás összeomlását okozzák, másrészt az adataikat is védeni kell egymástól. Több felhasználós rendszerben a gépet használók tárolt és éppen használatban lévő adatait is meg kell védeni a többiek általi hozzáféréstől vagy módosítástól. Az operációs rendszernek meg kell védenie használóit a hálózaton keresztül (többnyire adatszerzési vagy más rosszindulatú célból) indított támadásoktól is.
7. Felhasználói felület: az operációs rendszernek az input és output eszközökön keresztül biztosítania kell, hogy a felhasználó arra használhassa a számítógépet, amire való: számára hasznos programokat futtathasson rajta, kommunikálhasson velük, hozzáférhessen adataihoz, stb. Két alapvető formája a karakteres parancsértelmező (pl. cmd.exe, bash) és az intuitív grafikus felület (Aero, KDE).
8. Folyamatok közötti kommunikáció: angolul IPC („*inter-process communication*”). Az operációs rendszernek biztosítania kell, hogy a különböző folyamatok (pontosabban a különböző, - nem feltétlenül egy folyamathoz tartozó – szálak) adatokat cserélhessenek egymással, annak ellenére hogy nem érhetik el egymás memóriaterületeit. Sokféle technikával megvalósítható, egy OS jellemzően többet is támogat. (Pl. üzenetküldés alapú IPC, szinkronizáció, osztott memória, távoli eljáráshívás, socket-ek, szignál stb.)

Van még jó néhány kisebb-nagyobb szolgáltatás, de ezek a legalapvetőbbek. Azért volt fontos őket itt felsorolni, mert a később vizsgált rendszerek alapvetően abban térnek el egymástól, hogy ezeket hol és milyen módon biztosítják.

Kernel alatt az operációs rendszer magját értjük, ami – többek között – az előzőleg felsorolt szolgáltatásokat nyújtja a felhasználói programok számára. (Kivétel ez alól a felhasználói felület, ami nem a programok számára fontos, hanem a felhasználó számára, és amit általában nem tekintünk a kernel részének, még akkor sem, ha a kernel space-ben fut. – Lásd lentebb!) Más megfogalmazás szerint a kernel a hardveres erőforrásokat kezelő része a rendszernek, a legalsó szoftveres absztrakciós réteg a „jéghideg" hardver felett. (Azaz lényegében a HAL, valamint a memória és a processzor kezeléséért felelős részek - lásd folyamatok). Míg a kernel, mint látjuk funkcionálisan elég jól körülhatárolható, a technikai megvalósítást tekintve már nehéz ilyen általános ismérveket találni rá, hiszen sokféleképpen nézhet ki az egy darab bináris modultól kezdve a mikrokerneles implementációig, ahol a szolgáltatások nagy része külön ún. szerver folyamatokba van szervezve, amik megjelenésükben nem is nagyon különböznek a felhasználói programok folyamataitól. (És szintén user-space-ben futnak.)

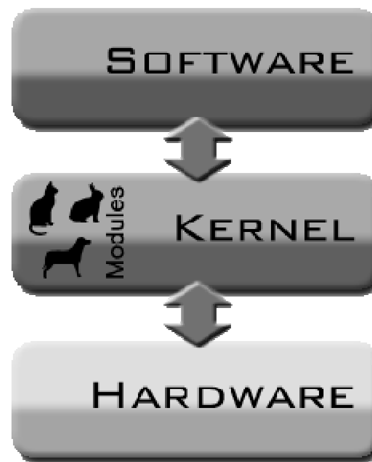
A kernel definíciója nem összekeverendő (de könnyen összekever*hető*) azzal, amit az angol szakterminológiában *kernel-space*-nek neveznek. Ez a tárban lévő végrehajtható kódnak az a része (vagy más nézőpontból az a tárrész), amely ténylegesen kommunikál a hardverrel, és evégett privilegizált módban fut, hogy a megfelelő inkriminált utasításait végre tudja hajtani. „Ellentéte" a *user space*.

Az ezután következő fejezeteket precíz megértéséhez még egy fogalom ismerete szükséges, ez pedig a kontextus-váltás: ez az a feladatsor melyet a kernel a folyamat-kezelés során végez el, mikor átvált az éppen futó folyamatról egy másikra. Ilyenkor számos dolgot el kell végeznie az operációs rendszernek: elsősorban ki kell mentenie a megszakított folyamat aktuális állapotára vonatkozó összes információt (processzor-regiszterek tartalma, operációs rendszer specifikus adatok, jelzőbitek stb.) hogy később ugyanonnan folytathassa a futást, és be kell töltenie/állítania a másik folyamatról korábban elmentett hasonló adatokat. Ez jellemzően egy számítás-, és így időigényes feladat. (A szálak közötti váltás általában jóval kevésbé időigényes, mint a folyamatok közötti). A kontextus-váltástól megkülönböztetendő a

„módváltás” amikor az eredeti folyamat marad az aktív, de a CPU átvált privilegizált módba, mivel az meghívott egy kernel-szolgáltatást, és annak a kódja fut.

2. b, A monolitikus kernel

A monolitikus kernel lényegében egyetlen nagy privilegizált módban futó bináris modul. Ez a modul tartalmazza az összes alapvető kernelszolgáltatás kódját. A monolitikus kernel lefordítva jellemzően egyetlen „futtatható” kimeneti fájlt ad. (Futtatható abban az értelemben, hogy a célprocesszoron végrehajtható gépi kódot tartalmaz, és nem úgy hogy egy másik operációs rendszer vagy akár „saját maga” alatt elindítható program lenne. Egy kernel tipikusan amúgy sem végez semmilyen „aktív” feladatot az után, hogy a rendszertöltő program betöltötte őt a tárba és elindította elkészíti az első valódi user-módú folyamatot, ami aztán valamilyen stratégia alapján sorban életre kelti a többi szolgáltatást, amiknek a rendszer konfigurációja szerint minden bootoláskor el kell indulniuk. Ide tartozik minden olyan program, ami nem alapvető operációs rendszer-szolgáltatást nyújt, de normál esetben mindig fut az adott rendszeren, mint például egy vírusirtó, vagy a grafikus felület ablakkezelője. Ezt a bizonyos „ős-folyamatot” a UNIX típusú rendszerekben init-nek hívják, az NT kernel alapú Windows-okban pedig smss.exe-nek. Ezek után a kernel már nem veszi át többet magának a processzort „önhatalmúlag”, csak a felhasználói folyamatok igényeit szolgálja ki. Természetesen azért előfordulhat, hogy visszakerül a végrehajtás a kernel-space-be anélkül, hogy bárki is egy rendszerhívást használt volna, de csakis esemény-vezérelt módon: például egy hardveres megszakítás hatására, vagy ha az egyik folyamat „rosszul viselkedik.” – Még egy megjegyzés: a fordítás utáni egy darab kimeneti fájl csak a modulárist támogatást nem tartalmazó, vagy legalábbis modulokat nem használó rendszerekre igaz. De erről majd egy kicsit később.)



A monolitikus kernelű rendszer felépítése

A monolitikus rendszerben tehát az egész kernel egyetlen bináris modul, ami tartalmazza a különböző szolgáltatásokat implementáló eljárásokat. Minden ilyen eljárás rendelkezik egy jól definiált interfésszel, amin keresztül meghívható. (Ezeket a definíciókat a Linux számára a POSIX szabvány írja elő.) Mivel itt az egész kernel gyakorlatilag egyetlen végrehajtható modul, ezért a monolitikus rendszer felépítését tekinthetjük úgy, hogy egyetlen „rendszer-folyamat" létezik, maga a kernel, és a felhasználói folyamatok minden rendszerszolgáltatást úgy érnek el, hogy ezzel a folyamattal kommunikálnak, rendszerhívások segítségével. (Monolitikus rendszereknél a kommunikáció a userland és a kernel között jellemzően csak rendszerhívások segítségével történik, melyek általában megszakításokkal vagy a még gyorsabb SYSENTER/SYSCALL – Intel/AMD – utasításokkal vannak implementálva. A hagyományos IPC a kernel és a felhasználói folyamatok között itt nem játszik szerepet.)

Az „egyetlen folyamat" megközelítésből már látszik az is, ami a monolitikus rendszereknek egyszerre a legnagyobb erőssége és gyengesége: az egész kernel, az összes szolgáltatás implementációja egy címtérben helyezkedik el. Miután valamelyik felhasználói folyamat végrehajt egy rendszerhívást, és a vezérlés átkerül az azt kezelő kernelbeli rutinhoz, a kernelen belül minden további szolgáltatás igénybevétele már csupán egy egyszerű eljáráshívás (ami assembly nyelven történő programozás esetén akár egy egyszerű JUMP utasítássá is optimalizálható. Nincs közte kontextus vagy módváltás, és az eredeti híváskor is

csak módváltás van. Ennek jelentősége majd az összehasonlító fejezetben fog látszani igazából.)

A monolitikus rendszerek talán legnagyobb előnye a sebességükben rejlik. Egy rendszerhívást kiszolgáló rutinnak nagyon gyakran van szüksége privilegizált erőforrások elérésére, még akkor is, ha maga a hívás látszólag nem közvetlenül ilyen szolgáltatást nyújt. Monolitikus rendszerben mivel az egész kernel supervisor módban fut, így nyilván mindegyik ilyen eljárás akár közvetlenül is elérhetné a hardvert, de a kernelt író programozó valószínűleg használni fogja a már előre megírt, (vagy mindenesetre futásidőben már rendelkezésre álló, és előre definiált interfészű) I/O eljárásokat, mivel egyrészt persze nem mazochista, másrészt lehetőleg minimalizálnia kell a kernel méretét. Mindenesetre ezeknek a kereszthívásoknak a száma a viszonylag kevés, hiszen a kernelen belül „mindenki tudja" mit hol keressen, és hozzá is fér, nincs szükség közvetítőkre. Ráadásul, ahogy láttuk, ezek a hívások nem járnak kontextusváltással és lassú folyamatközi kommunikációval.

A monolitikus kernel felépítése fejlesztői szempontból vitára szokott okot adni. Van, aki szerint az alapvetően strukturálatlan szerkezete miatt nehezebb implementálni, a forráskódját átlátni, karban tartani. (Bár jegyezzük itt meg, hogy azért a monolitikus kernelbe is lehet struktúrát vinni, valójában nem kötelező „ömlesztve" deklarálni az összes eljárást és függvényt, sőt ésszerű tervezés mellett szinte törvényszerű hogy külön, jól definiált belépési pontú struktúrákba legyenek szervezve. Ettől technikailag még egyetlen összefüggő címtérben futó kernelünk lesz.) Más érvek szerint a monolitikus kernel éppen hogy egyszerűbb, mivel nem emel mesterségesen és fölöslegesen (?) határokat és közvetítőket a szolgáltatások és adatstruktúrák közé. (Ez utóbbi álláspontot osztják olyan szaktekintélyek is az operációs rendszer fejlesztés területéről, mint Linus Torvalds és Ken Thompson.)

Egy másik elméleti jellegű érv lehet a monolitikus rendszerfelépítés mellett, hogy a szolgáltatások közötti közvetlen kommunikáció, a kevesebb „töltelékkód" miatt egyrészt kisebb lesz az egész kernel mérete, (most nem a kernel-space-ben futó kód méretéről van szó, hanem az összes alapvető szolgáltatást nyújtó kód összméretéről!) ami egyrészt ugye erőforrás-takarékos, másrészt kisebb forráskódban statisztikailag is kevesebb a hiba. (Viszont az is igaz hogy egy esetleg felbukkanó bug sokkal nagyobb, - kb. 100%-os – valószínűséggel fog a sokkal veszélyesebb kernel space-ben végrehajtódni!)

Még mindig a fejlesztési szempontokat vizsgálva vannak ennek a modellnek valóban cáfolhatatlan hátrányai is. Új funkcionalitás hozzáadása a kernelhez igen macerás, általában az egész újrafordításával jár, ami (persze az adott operációs rendszertől, a hardvertől, és még sok minden egyébtől függően) igen idő- és erőforrás-igényes lehet. Egyetlen apró hiba kijavításához szintén szükség lehet a kernel újrafordítására, és a gép újraindítására. És akkor még nem is beszéltünk a tesztelés nehézségeiről.

Portolhatóság szempontjából elvileg szintén nem túl jó a modell, hiszen a hardvert közvetlenül használó, arra támaszkodó modul mérete nagy. Más architektúrára való átköltöztetéskor potenciálisan az egész kódbázist módosítani kellhet, hiszen a HAL nincsen élesen elválasztva a kernel többi részétől. Ugyanakkor a gyakorlat ennek az elvnek ellentmondani látszik, hiszen minden idők talán két legtöbb architektúrára átportolt rendszere, a UNIX és a Linux is monolitikus felépítésű.

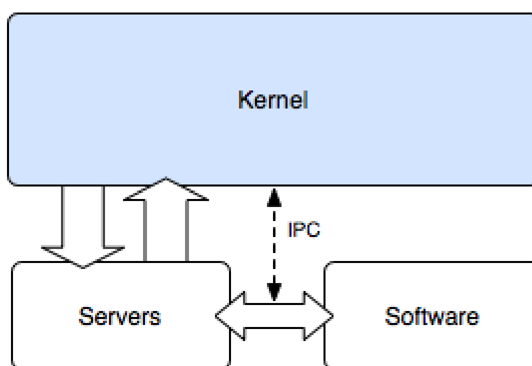
A monolitikus rendszerekkel szemben leggyakrabban felhozott érvek azonban nem ezek közül egy, hanem valami más - és szintén tagadhatatlan: mivel az egész kernel egy címtérben fut, benne egyetlen bug is a teljes rendszer összeomlását okozhatja. Az még hagyján hogy mint fentebb írtam egy kritikus hiba sokkal nagyobb valószínűséggel fog kernel-spaceben bekövetkezni, mivel az ott futó kód mérete nagy, de a hibás szolgáltatás a közös címtérnek köszönhetően a teljes kernelt „földre viheti”, hiszen a többiek állapotát is módosíthatja, hozzáfér az adatstruktúráikhoz, kódjukhoz stb.

Végezetül említsük még meg a monolitikus rendszereknek egy speciális változatát, a betölthető kernelmodul-támogatással rendelkező operációs rendszereket. (Speciális változatot írtam, de nem indokolt, lévén ma már szinte minden modern monolitikus rendszer rendelkezik ezzel a képességgel – mint például a Linux vagy a Solaris is.) Ez alapvetően azt jelenti, hogy a kernel fordításakor bizonyos részek nem a „fő”, a kernel nagy részét tartalmazó kimeneti fájlba kerülnek, hanem egy külön bináris fájlba. Ezeket aztán később, futási időben képes a kernel betölteni és beilleszteni a memóriában lévő kernel-képbe, mintha eredetileg is egybe lettek volna fordítva, aztán ha a később már ismét nincs szükség az adott részre, el is távolíthatja tárból. Ily módon szükség esetén dinamikusan, újrafordítás, és újraindítás nélkül kibővíthetők a kernel képességei. Ez azért hasznos, mert így a ritkán használt szolgáltatások (jellemzően például eszközmeghajtók, vagy bizonyos fájl-rendszerek moduljai) kint tarthatók a kernelből, így minimalizálva annak méretét anélkül, hogy le kellene mondani az adott

funkcionalitásról. Ez még azt az árat is megéri hogy amikor mégis szükség van az adott szolgáltatásra először meg kell várni míg a megfelelő modul betöltésre kerül. Ez a képesség nem összekeverendő a mikrokernelek strukturális szintű modularitásával!

2. c, A mikrokernel

A mikrokerneles rendszerek alapvetően egy másfajta filozófiára épülnek, mint a monolitikus modellt követőek. Egy operációs rendszer, szűkebben véve a kernel feladata lenne megvédeni egy egész rendszer stabilitását a felhasználói programok hibáitól, méghozzá úgy hogy hardveres támogatást kihasználva csak a saját kódját futtatja a kritikus privilegizált módban. Ugyanakkor, ha belegondolunk abba hogy e mögött a bináris kód mögött a Linux esetében pl. 2.500.000 sornyi forráskód áll, míg a Windows NT kernel mögött durván 5.000.000 sor, óhatatlanul felmerül a kérdés hogy elképzelhető-e hogy ezekben magukban nincsen jó néhány kritikus hiba? A válasz természetesen nem. (Még akkor is, ha figyelembesszük hogy pl. egy átlagos lefordított bináris Linux kernel-képbe a teljes forráskódnak csak kis része „kerül bele”.) A mikrokernel mögötti alapelv az, hogy próbáljunk minél kevesebb kódot, csak a feltétlenül szükséges szolgáltatások implementációját beletenni a privilegizált módban futó hagyományos értelemben vett kernelbe, és minden más szolgáltatást külön-külön folyamatokkal, a user space-ben valósítsunk meg.



A mikrokerneles rendszer felépítése

Ha az alapvető fejlődési irányt nézzük, akkor elmondhatjuk, hogy a mikrokerneles rendszereknek sikerült ezt a célt egyre jobban megvalósítani. Konkrét példákba majd az ezzel foglalkozó fejezetben mennék bele, most csak annyit, hogy míg a modell ősatyjának számító Mach kernelbe bizony még sok „lim-lom” belekerült, addig az új, második generációs mikrokernelekben, mint az L4, már sikerült ezt néhány tényleg minimális szolgáltatásra

leszűkíteni, mint a címtér-kezelés, szálkezelés, IPC-kezelés, időzítő-kezelés, és egy alapvető HAL szolgáltatás. (Az L4 kernel mérete mindössze 12 kB és összesen 7 rendszerhívást szolgál ki.) Ezekből a kiszolgálórutinokból egy a monolitikus rendszerekéhez hasonló (de persze sokkal kisebb) hagyományos értelemben vett, a kernel space-ben, privilegizált módban futó kernel épül, amit a felhasználó és a szerver-folyamatok is normál rendszerhívásokkal érhetnek el. Az összes többi rendszerszolgáltatás azonban normál felhasználói folyamatként lesz implementálva, és persze nem egyetlen folyamatban, hanem minden egyes szolgáltatás (vagy inkább szolgáltatás-család, mint például hálózati protokollok, fájlrendszerek, egyes eszközök eszközmeghajtói, stb.) a maga külön folyamatában, hiszen e nélkül elveszne a modell egyik lényege. Ezek az ún. szerver-folyamatok aztán egymással és a felhasználói folyamatokkal az egyes rendszerekben különbözőféleképpen implementált IPC hívások segítségével kommunikálnak. (A szerver-folyamatok időnként azért különbözhetnek egy kissé a valódi felhasználói folyamatoktól, ugyanis előfordul, hogy a kerneltől különleges hozzáférési jogokat kapnak bizonyos erőforrásokhoz, főleg bizonyos memóriaterületekhez, hogy ne kelljen mindig a mikrokernelhez fordulniuk emiatt.)

A modell legnagyobb előnyeként általában a kiemelkedő megbízhatóságot említik a támogatói. Ez egyrészt annak köszönhető, hogy a kritikus, privilegizált módban futó kód mérete a minimálisra lett csökkentve. Ekkora forráskódot már reálisan lehetséges hibamentesen tartani. (Meg persze kevésbé változik is a fejlesztés során.) Másrészt, és ez talán még nagyobb vívmány, ott van a szerverek kicserélhetősége. Egy monolitikus rendszerben ha egy alapvető rendszerhívás kiszolgáló rutinja működésképtelenné válik akkor az nagyon jó eséllyel a rendszer összeomlásához és újraindításhoz vezet még akkor is, ha a többiben nem okozott hibát, hiszen az általa biztosított szolgáltatás nélkül a rendszer működésképtelen volt. Egy mikrokerneles rendszerben ilyenkor az adott szervert egyszerűen leállítják, és indítanak belőle egy másik példányt. Ráadásul mivel minden szerver és a mikrokernel is külön címtérben van, nem okozhatnak közvetlen módon olyan hibákat egymásban, mint egy monolitikus rendszer esetében. (A gyakorlatban ez persze nem megy ilyen karikacsapás-szerűen, mivel a szerverek azért támaszkodnak egymás szolgáltatásaira, így amikor az egyik összeomlik hajlamos magával rántani a többit is, annak ellenére, hogy külön címtérben vannak. Néhány modernebb mikrokerneles rendszerben azonban már vannak többé-kevésbé működő megoldások erre a problémára, nevezetesen minden szerver köteles

folyamatosan egy megadott helyre „menteni” az állapotát, és leállás esetén a rá támaszkodó más szerverek felfüggesztett állapotba kerülnek addig, amíg egy új példány el nem indul belőle, és vissza nem állítja az előző állapotát a mentés alapján.)

A mikrokerneles modell hívei szerint az ilyen rendszer strukturáltabb, logikusabb, áttekinthetőbb, egyszerűbb (?) és szerkezete megbízhatóbb, kevesebb hibát tartalmazó operációs rendszert eredményez. (Persze ezzel nem mindenki ért egyet, de erről már volt szó az előző alfejezetben.)

Az mindenesetre elvitathatatlan, hogy bár sokak szerint a monolitikus felépítés közelebb áll az emberi gondolkodáshoz és kevesebb munkával implementálható, addig a fejlesztés gyakorlati része, főleg a hibajavítás és a tesztelés jelentősen könnyebb egy mikrokerneles rendszerben. (Kivéve persze, ha magát a mikrokernelt kell módosítani, de erre igen ritkán van szükség.) Egy szerver kicserélése igen egyszerű, sőt azt is megtehetjük, hogy egy ideig párhuzamosan futtatjuk az új verziót a régivel tesztelés céljából. Hasonlóképpen új funkcionalitás hozzáadása a rendszerhez is igen rugalmasan megoldható.

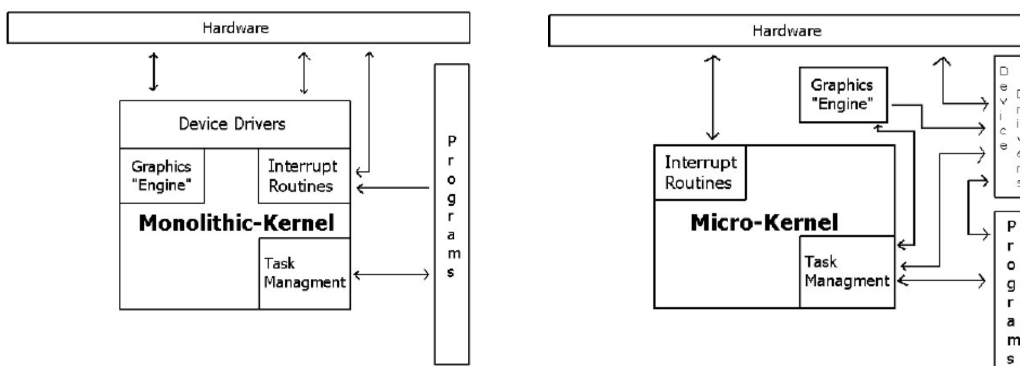
Természetesen ennek a modellnek is megvannak a maga hátrányai. Egyrészt általában nagyobb az összesített memóriaigénye, mint egy hasonló szolgáltatásokat nyújtó monolitikus rendszernek. Ráadásul bizonyos feladatokat, mint pl. a lapozási stratégiák, és a folyamatütemezés nem vagy csak igen nehezen lehet hatékonyan implementálni user space folyamatokban. A legnagyobb kifogások persze nem ezek, hanem egy sokkal alapvetőbb: a teljesítmény kérdése. Az első gyakorlatban is kipróbált Mach3 kernellel felszerelt UNIX-ok képesek voltak akár 50%-kal is alulteljesíteni monolitikus társaikkal szemben. Ez a teljesítménycsökkenés főleg a sokkal több kontextusváltásnak, és a közvetlen eljáráshívásnál lassabb IPC-nek volt köszönhető. A mai mikrokerneles rendszerek ennél már jóval kedvezőbb teljesítményre képesek, bár többnyire még mindig nem érik el egyes monolitikus társaikét azonos platformon. Ami igazán meglepő lehet az a megoldás: az L4 például a Mach3-hoz képest még több szolgáltatást kimozgatott user space-be, ami pedig még több kontextusváltást okoz – elméletben legalábbis. Ugyanis a gyakorlatban az L4 jobban teljesít, mivel a korábbi rendszerekben az IPC-vel kapcsolatos „elvesztegetett” CPU-ciklusok legjelentősebb része az üzenetekhez kapcsolódó jogosultságellenőrzés volt, ami a Mach3-ban például még a mikrokernel része volt, amit az lelkiismeretesen végre is hajtott minden egyes üzenetnél.

Mostanra azonban már ez is ki lett helyezve egy külön saját folyamatba. Ráadásul az L4-ben alkotója, Jochen Liedtke magát az IPC algoritmust is jelentősen egyszerűsítette.

Még egy érdekes lehetőség adódik abból hogy egy mikrokerneles rendszerben az API-t szolgáltató eljárások nagy része külön felhasználói módú folyamatokban fut. Ugyanis így elvileg (és gyakorlatilag is) megoldható egy mikrokernel fölött több API, virtuálisan több operációs rendszer futtatása is, a fentitől eltekintve teljesítménycsökkenés nélkül. Sőt, valójában lehetséges a mikrokernel fölött egy mesterségesen megkonstruált „emulált" rendszerhívás interfész helyett egy komplett monolitikus kernel futtatása is user space-ben, csupán egy kicsit módosítani kell azon, hogy ne közvetlenül a hardvert, hanem a mikrokernel által biztosított absztrakciós réteget használja.

2. d, Alapvető különbségek és hatásaik

Ahogy az előző fejezetekben jól látszott hogy szerkezetileg a fő különbség a két modell között az hogy a rendszer szolgáltatásokat implementáló kód mekkora része fut privilegizált módban, illetve hogy ezek a szolgáltatások hogy vannak magának a rendszernek a szempontjából elkülönítve. (Értsd ez alatt: megosztják- e egymással címterüket, illetve a CPU-ért konkuráló külön folyamatként futnak-e stb.) A szerkezeti különbségből természetesen működésbeli különbség is adódik, ami lényegében abban mutatkozik meg, hogy hogyan kommunikálnak a felhasználói folyamatok az operációs rendszerrel, illetve hogyan kommunikálnak egymással az operációs rendszer részei. (És itt most ismét csak a szorosan vett alapvető rendszerszolgáltatásokat értjük ez alatt, hiszen hiába szállítanak nekünk egy szövegszerkesztőt az operációs rendszerünkkel „egybecsomagolva", attól az még egy sima felhasználói folyamat lesz a rendszer szempontjából.)



Eltérő szerkezet, eltérő működés – a két rendszermodell

Természetesen mindkét modellt követve felépíthetünk egy minden modern igénynek megfelelő, korszerű operációs rendszert. Mint ahogy azt a bevezetőben felsorolt példák is alátámasztják, mindkét „iskola" alapján készíthetünk olyan rendszert, melyben bármilyen általános vagy speciális célú felhasználó program futtatható, legalábbis a hardver által meghatározott korlátok között. Nincs semmilyen elvárható szolgáltatás, funkció vagy alapvető tulajdonság, amivel csakis az egyik módon felépített rendszer rendelkezhetne. Ez viszont természetesen egyáltalán nem jelenti azt, hogy gyakorlati szempontból a két modell teljesen egyenértékű lenne, azaz mindegy volna melyik mentén kezdjük el operációs rendszerünket felépíteni. (Ebben a fejezetben csupán a felhasználók szempontjából vizsgálom

a különbségeket, azaz azokat az attribútumokat veszem górcső alá, melyek az operációs rendszer „naiv” felhasználói számára is érzékelhetőek és jelentőséggel bírnak. Persze ha valóban el akarunk játszani a gondolattal hogy operációs rendszert írunk, akkor egyéb tényezőket is érdemes figyelembe venni a választásnál, ugyanis a két modell alkalmazásának a fejlesztés során is fel fognak merülni a maguk igencsak kézzelfogható előnyei és hátrányai. Általánosságban elmondhatjuk hogy míg lokális szinten a mikrokerneles felépítés átláthatóbb kódot eredményez, addig globálisan bonyolultabb és nehezebben áttekinthető struktúrájú programunk lesz, hiszen az egyes modulok kommunikációjánál számolnunk kell az IPC megvalósításának körülményességével is. A gyakorlat azt mutatja hogy ha egy fejlesztő szabadon választhat akkor gyakrabban fogja a monolitikus szerkezet mellett letenni a voksát. Ez egyrészt kevesebb kódolási és tervezési munkát jelent általában, másrészről ne felejtsük el hogy egy kód olyan amilyenné írjuk – tehát egy monolitikus kernel forráskódja is lehet remekül strukturált és kezelhető ha figyelmet fordítunk rá. Ennek a tézisnek az egyik legnagyobb szószólója maga Linus Torvalds, aki hosszú évek óta vitatja a fentebb megfogalmazott állítást, miszerint a mikrokerneles forma kódszinten könnyebben kezelhető volna. Az hozzáértését a témához nehéz volna kétségbe vonni, úgyhogy maradjunk annyiban hogy a mikrokerneles modell sokkal inkább „kikényszeríti” a strukturált kódot.)

Felhasználói szempontból mindez azonban nem számít, csak a végeredmény. (Azért nyilván a forrás bonyolultsági szintje adott programozó „mellett” igencsak befolyásolja, hogy végül milyen futtatható kód lesz az eredmény, de az ilyen közvetetett és nehezen mérhető és számszerűsíthető hatásokkal itt most ne foglalkozzunk.). Ha „ugyanazt” a rendszert mindkét módszerrel megpróbáljuk megírni a végső formában több mint valószínű, hogy csak két érzékelhető és alapvető fontosságú különbséget fogunk találni: a teljesítményük közöttit, (teljesítmény alatt értsd: sebesség,) illetve a stabilitás bélit. (Ha nem tiszta mit értek ezek alatt, akkor nézzük meg a hagyományos Torvalds-féle Linux kernelt, és az abból készült L4Linuxot. Ez utóbbi egy normál Linux kernel annyi módosítással hogy a moduljai külön folyamatban futnak, és a hardver absztrakciós rétege nem közvetlenül a BIOS-t éri el, hanem egy alatta futó L4 mikrokernel rendszerhívásait használja. Lesz még róluk szó bővebben ebben a fejezetben, ugyanis ők lesznek a két gyakorlatban is implementált operációs rendszer melyeket felhasználok példaként az összehasonlításhoz. A két technológia praktikus szempontok szerinti (sebesség, hibatűrés és gyakoriság) összehasonlításának csak úgy van

értelme, ha két rendszer szolgáltatásai algoritmus és kód szerint is megegyeznek a két modell közötti különbségekből adódó eltéréseket leszámítva. (Persze összehasonlíthatnánk egy mobiltelefonokon futó, és alig néhány szolgáltatást nyújtó Symbian-t is egy szuperszerverekre tervezett BSD Unix-szal, de túl sok érdemi következtetést nem lehet belőlük levonni. Ez persze extrém példa volt, de könnyen beláthatjuk, hogy még két azonos kategóriába tartozó operációs rendszer esetén sem lenne értelme az összevetésnek úgy, ha nem tudjuk biztosítani, hogy a vizsgált rendszerekben az egyes szolgáltatások algoritmusai relatíve milyen hatékonysággal oldják meg a kitűzött feladatukat, illetve milyen gyakorisággal tartalmaznak és milyen súlyos hibákat. Így ugyanis nem tudhatnánk, hogy a mért eredmények közötti eltérések mennyiben írhatóak az alapvetően eltérő tervezési modellek számlájára, és mennyiben a kódok „minőségében" meglévő különbségekére. Tehát a „hagyományos" és az L4Linux-nál jobb rendszereket nehezen találhatnánk az összehasonlításhoz.)

Az előző fejezetekben ismertetve lett tehát a két kernelmodell közötti különbség a strukturális felépítés és általános működés tekintetében. Most azonban vizsgáljuk meg hogy milyen különbségeket okoznak ezek a rendszer mindennapi felhasználás szempontjából fontos tulajdonságaiban. Két ilyen tulajdonságot szokás a témával kapcsolatban felhozni, ezek pedig a stabilitás (más terminológiával nevezhetjük biztonságnak is, bár ennek a szónak a használata gyakran vezet félreértésekhez, mivel több különböző jelentésű angol kifejezésnek is ez az egyetlen használható magyar megfelelője,) illetve a teljesítmény (sebesség.) Kezdjük most az elsővel.

Mielőtt nagyon belemerülnénk a stabilitás kérdésébe, egy dolgot nem árt tisztázni, mégpedig azt hogy mit is értünk alatta: a rendszer ellenálló képességét a hibákkal szemben. Tehát a stabil rendszer nem egyenlő a hibamentes rendszerrel, mivel olyan – egy programozók körében közismert humoros „axióma" szerint – nem létezik. Hiba minden kódban van tehát, az hogy mennyi csupán az azt készítő programozó(k) hozzáértésétől és tapasztalatától függ. (Na meg persze a kód hosszától. A tökéletes rendszer létrehozhatatlanságának paradoxonját (?) jobban megérthetjük, ha átgondoljuk hogy egy hiba javítása maga is kódmódosítással és új kód írásával jár, ami statisztikailag új hibák keletkezését jelenti.) Egy stabil rendszernek viszont minimalizálnia kell a bekövetkező hibákból eredő károkat. És ha „rendszer" alatt operációs rendszert értünk akkor a kérdés mindjárt még komolyabb lesz, hiszen egy operációs rendszernek nem csak a saját hibáit kell

„lekezelnie”, de meg kell védeni saját magát, a felhasználót és annak adatait a rajta futó felhasználói programok hibáitól is. Itt függ össze a stabilitás és a biztonság kérdése is: egy tökéletes világban senki sem lenne képes elolvasni, módosítani vagy törölni mások adatait, vagy leállítani, hibás működésre késztetni mások programjait, mert minden program úgy működne, hogy csak a megfelelő jogokkal rendelkező személy (vagy program) kérésére lenne hajlandó végrehajtani ezeket az érzékeny műveleteket. Ezek persze a valóságban is célkitűzések minden professzionális hozzáállással fejlesztett szoftver írásánál, de valóság az hogy ennek ellenére programjaink ma (és a belátható jövőben ez nem is látszik változni) nem ilyen biztonságosak. Ha manapság ha arról hallunk, hogy valamilyen támadás érte a gépünket, akkor biztosak lehetünk benne, hogy ez nem jelent mást, mint hogy valaki vagy valami szándékosan megpróbálja kihasználni, előidézni szoftvereinkben (gyakran magában az operációs rendszerben) meglévő hibákat – általában károkozási, adatszerzési szándékkal. Ha a számítógépünk vírussal fertőződik, egy hacker a keresztül keresztül feltöri, vagy bizalmas adataink egy meglátogatott weblapot tároló szerverre kerülnek, akkor szinte biztosak lehetünk benne, hogy a használt szoftvereinkben használtak ki valamilyen hibát, pl. egy nem kezelt veremtúlcsordulást vagy egy elmulasztott jogosultságellenőrzést. (Ezek a hibák tehát ugyanúgy lehetnek például a böngészőnk, levelezőprogramunk kódjában mint magában az operációs rendszerében - de a kettő következményei általában nem azonos súllyal esnek a latba, mint ahogy azt látni fogjuk.) Ebből kifolyólag mikor a rendszerünk stabilitásáról beszélünk akkor egyben a biztonságáról is beszélünk.

Egy operációs rendszer természetesen nem tudja megakadályozni, hogy a felhasználói programokban hibák legyenek, mint ahogy azzal sem számolhat, hogy saját magában ne lennének. Viszont feladata ezekez a hibákat minél jobban izolálni, azaz káros következményeiket minimalizálni, lehetőleg kiküszöbölni. Ebbéli képessége pedig nagyban függ a választott kernelmodelltől.

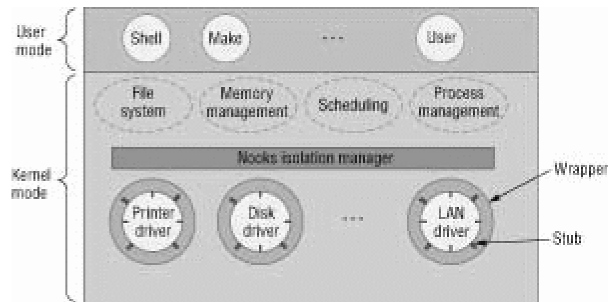
Ahogy már a szóhasználatból is kiderült („izolálni”) a probléma megoldása a hibás kód hozzáférésének korlátozása, minimalizálása a többi folyamat memóriaterületeihez, a többiek adatstruktúráihoz illetve a hardverhez, mivel ezek „segítségével” tud egy program a saját működési területén túlnyúló károkat okozni. Függetlenül attól melyik modellre épül a rendszerünk az egyszerű felhasználói programok a saját külön pfolyamatokban futnak – külön címtérben, korlátozott hozzáférési módban. Ennek megfelelően az elv az, hogy nem férhetnek

hozzá mások memóriaterületeihez, mint ahogy a hardver erőforrásokhoz sem, azaz csak a saját „munkájukat” képesek tönkretenni, illetve a saját adataikat képesek illetéktelenekhez továbbítani. Ez még mindig nem túl örömteli, de az ilyen hibát operációs rendszer szinten megakadályozni nem lehet. (Az utóbb említettet megkísérelni azért igen, hiszen az adattovábbítás általában az operációs rendszer által kontrollált hálózati interfészeken keresztül történik. Csakhogy a rendszer számára kiszűrni, hogy mi lehet a legálisan továbbküldhető adat és mi nem legfeljebb a címzett alapján sikerülhet, tartalom alapján aligha, hiszen ő nem ismeri az adatok jelentését – ilyen szintű védelem már komoly mesterséges intelligenciát feltételezne az operációs rendszer részéről, és bár nem lehet kizárni hogy egyszer majd készülnek ilyen rendszerek, ez nem témája ennek a dolgozatnak.) A fent említett korlátozás alapú védelem hardveres alapú (az MMU illetve a védett módú CPU valósítja meg,) így arról feltételezhetjük (illetve nagyjából kénytelenek vagyunk) hogy tökéletesen működik. Egy felhasználói program tehát közvetlenül nem érhet el érzékeny erőforrásokat, rendszer szintű károkat csak úgy okozhat ha - véletlenül, vagy szándékosan – hibásan működő operációs rendszer-szolgáltatásokat hív meg. Ezeknek a kivédése tehát az igazán jelentős kihívás, és ennek kapcsán beszélhetünk az operációs rendszer stabilitásáról – szemben egy felhasználói program stabilitásával.

A hagyományos monolitikus rendszerekben a kernelspace-ben fellépő hibák kontrollálása futásidőben igen nehéz, sőt többnyire lehetetlen feladat. A hibás kódnak szabad hozzáférése van a hardverhez és az összes memóriaterülethez, azaz a gépen lévő összes adathoz és erőforráshoz. Ezekkel szabadon visszaélhet, és ráadásul mivel egy címtérben fut az egész kernel az összes többi szolgáltatáshoz is hozzáférhet és elronthatja őket. (Ez alatt most „véletlen” elrontást értettem, ami könnyen előfordulhat, hiszen a kernelen belül az egyes szubrutinok egymásra támaszkodnak, egy hibája az őt használó többiben is hibákat okozhat. Természetesen az is egy felmerülő lehetőség hogy a hibás kódrész szándékosan módosítja kedvezőtlenül a többi modult is, azaz hogy egy eleve ártó szándékkal megírt algoritmussal van dolgunk. Nyilván normális esetben ilyen nem fordulhatna elő egy széleskörű használatra szánt rendszerben, de a gyakorlatban meg szokott történni. Persze nem úgy hogy a felhasználó vagy a rendszeradminisztrátor véletlenül bennfelejti egy vírus forráskódját a kernelében és belefordítja, hanem egy eredetileg felhasználói módban futó kódrészlet egy hibásan működő rendszerhívás segítségével „bemásoltatja” magát valamilyen privilegizált

térben lévő memóriacímre, és miután rá kerül a vezérlés a „megfertőzött” rendszerszolgáltatás hívása útján már szabadon másolgathatja magát a kernelen belül. Az ilyen jellegű kártevő programok működése gondos jogosultságellenőrzéssel, illetve a kernel módban futó szolgáltatások lehetőség szerinti hibamenetesen tartásával nehezíthető meg. Ez utóbbit tökéletesen fenntartani természetesen a lehetetlennel határos egy modern monolitikus rendszerben.

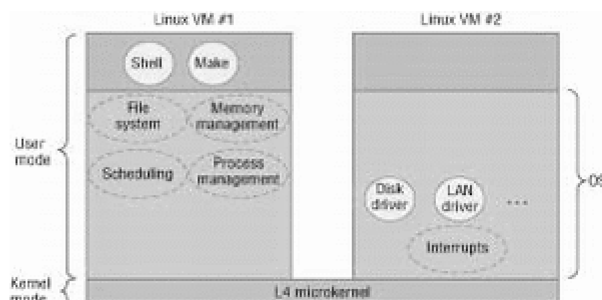
Ezeknek a hátrányoknak a kiküszöbölésére számos többé-kevésbé hatékony megoldás és javaslat született az elmúlt években, évtizedekben. Az egyik ilyen a „Nooks” technológia, ami a monolitikus kernelen belül az eszközmeghajtók köré egy-egy speciális wrapper (csomagoló) kódot helyez, és a driver csak azzal kommunikálhat, és az továbbítja a kéréseit a kernel többi része felé. (A kitüntetett figyelem nem véletlen, ugyanis ezek az illesztőprogramok amellett, hogy a privilegizált módban futtatott kódnak igen jelentős hányadát teszik ki és sokszor külső forrásból, a hardver gyártójától származnak, ráadásul hagyományos a legtöbb súlyos hibát is tartalmazzák – ez valószínűleg bonyolult feladatukból következik.) Ezeknek a csomagoló-rétegeknek a feladata kettős: egyrészt védik a kernelt a driver hibás hívásaitól, másrészt amennyiben az összeomlik, újraindítja. Hiszen egy eszközmeghajtó úgy is működésképtelenné teheti a rendszert, ha nem nyúl semmi máshoz, egyszerűen beszünteti a működését és többet nem lesz elérhető a hardver eszköz amihez tartozott. A Nooks-szal jelentősen javítható egy monolitikus rendszer stabilitása, anélkül hogy a kernelen drasztikusan módosítani kellene. Azonban megvannak maga korlátai is: a driver hardver-elérését továbbra sem ellenőrzi semmi, mint ahogy a memória izoláció sem működik igazán, lévén az eszközmeghajtó még mindig privilegizált módban fut. Ráadásul egy nooks-os wrapper nem biztos hogy észreveszi ha a driver működésképtelenné vált, arról nem is beszélve hogy magának a csomagolónak a kódjában is lehetnek hibák.



A „Nooks”

Egy másik, a „tökéletességhez” sokkal jobban közelítő megoldás a virtualizáció. Itt egy ú.n. „*virtual monitor*” program fut közvetlenül a valódi hardveren, és annak erőforrásait megosztva virtuális számítógépeket hoz létre. A virtuális gépeken külön operációs rendszerek futhatnak, amik a hardver helyett a „virtuális hardvert”, a virtuális gép által nyújtott interfészt kezelik. Ily módon az egymástól megvédeni kívánt folyamatok a lehető legtávolabb kerülhetnek egymástól, mégpedig külön virtuális gépre, aminek következtében még csak nem is tudnak egymás létezéséről, még akkor sem ha „elvileg” kernel módban futnak. Ennek persze megvan a maga ára, ami egyrészt teljesítménycsökkenésben mutatkozik meg (a virtuális gépek összesen kevesebb erőforrás felett rendelkezhetnek, mint amennyit a valódi hardver biztosít, lévén a virtual monitor program is felhasznál azokból,) másrészt a ma elterjedt processzorok még nem képesek az operációs rendszer kódjának minden részét virtualizált módon futtatni, azaz a legtöbb operációs rendszert először módosítani kell hogy fusson a virtuális gépen. Ezt hívjuk paravirtualizációnak. (E dolgozat írásának időpontjában a világ két legnagyobb processzorgyártója, az Intek és az AMD az igények kielégítésére már előállt a korábbiaknál sokkal komolyabb virtualizációs támogatást biztosító modellekkel, ami a technológia előretörését jelzi. Ezeken a processzorokon - elvben – már lehet módosítás nélkül mindenféle operációs rendszert párhuzamosan futtatni – ez a valódi virtualizáció.) Napjaink legelterjedtebb virtualizációs szoftvercsomagjai közé tartoznak a Microsoft Virtual Machine, a VmWare vagy a nyílt forráskódú Bochs és Qemu. Bizonyos szempontból tekinthető virtuális gépes megoldásnak az ebben a fejezetben majd később tárgyalt L4Linux is, hiszen az L4 mikrokernel hasonlítható egy minimalista virtual monitorhoz, amin a módosított Linux kernelek futnak. (Igen, esetenként több is, ugyanis az összekapcsolást megvalósító kutatók hamar rájöttek, hogy a felhasználó programok szempontjából jóval stabilabb környezetet lehet teremteni ha a veszélyes drivereket egy külön userland

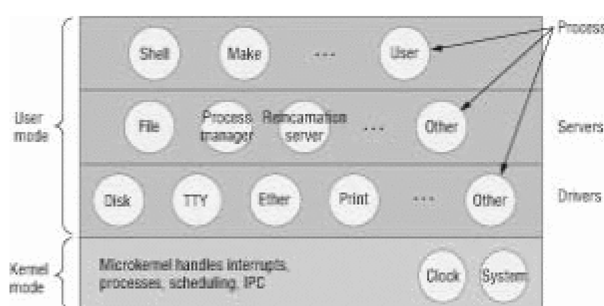
folyamatban futó kernelben, egy külön „virtuális gépben” helyezik el. Így ha valamelyik fatális hibát követne is el, csak a saját kernelét teheti tönkre, a felhasználói programokét nem, így elég azt újraindítani.) Minden estre én itt az L4Linux-ot nem így, hanem mint a hagyományos Linux egy speciális, mikrokerneles változatát fogom tárgyalni.



A több Linux példányt futtató L4 rendszer felépítése

A harmadik lehetséges megoldás az úgynevezett multiszerveres architektúra. Ez tulajdonképpen a hagyományos értelemben vett mikrokerneles modell. Nem húzunk a kernel köré egy mesterséges határoló réteget, ami kezeli helyette a hardvert (és amiben persze szintén lehetnek hibák) hanem hagyjuk azt közvetlenül elérni ezeket az erőforrásokat, de gondoskodunk róla hogy a kernel csak azt a kódot tartalmazza amelynek valóban elengedhetetlenül szükséges ez a hozzáférés. Azaz csak egy nagyon kisméretű kernelünk lesz, amibe csak a saját nagyon alaposan ellenőrzött kódunkat engedjük be. Ezzel a megoldással természetesen drasztikusan (sok nagyságrenddel) csökkenthetjük a privilegizált módban bekövetkező hibák számát. Az összes többi operációs rendszerszolgáltatás (így például a különösen veszélyes driverek is) közönséges „felhasználói” folyamatokba lesznek kimozgatva. (Az idézőjel nem véletlen, hiszen valamiféle csoportosítás így is létezik a userland folyamatok között aszerint hogy milyen szerepet töltenek be a rendszerben: a legalsó szinten a driverek vannak, ők többnyire csak a mikrokernelhez fordulnak kérelmekkel hogy elérhessék a rájuk bízott hardvert és kiszolgálhassák a hozzájuk a többi folyamattól beérkező kérelmeket. Rájuk épülnek az egyéb operációs rendszerszolgáltatásokat implementáló szerverek, mint például a hálózati protokollokat kezelők, a fájlrendszerek szerverei stb. Ezek „fölött” pedig már a valóban a felhasználói programok folyamatai futnak. Ezek teljesen hasonlóan működnek a monolitikus rendszerekre fejlesztett programokhoz, annyi különbséggel, hogy az I/O-hoz és egyéb rendszerszolgáltatásokhoz való hozzáférést

megvalósító könyvtári függvények, melyeket a programok felhasználnak, nem a hagyományos megszakításokkal, SYSCALL hívásokkal stb. operálnak, hanem a külön folyamatként futó szerverekkel kommunikálnak az adott rendszerben használt IPC-n keresztül. Arra is van lehetőség, hogy a mikrokernel egy táblázatban nyilvántartsa mely folyamatoknak mely erőforrásokhoz van joguk hozzáférést kérni. Tehát pl. I/O-t csak a driverek kezdeményezhetnek, és csak a saját hardver egységük irányába. Ez a megoldás tovább növeli a biztonságot, de nem minden mikrokerneles rendszerben van megvalósítva.)



A Minix mint klasszikus multiszerveres mikrokerneles rendszer

Tehát mint azt már a korábbi fejezetben is látszott a mikrokerneles rendszer megnövekedett stabilitása és biztonsága abban rejlik, hogy a valóban privilegizált módban futó kódrészlet, a kernel, igen kicsi, könnyen verifikálható és nem nagyon változik. Az összes többi rendszerszolgáltatás külön folyamatokban, a *szerverekben* fut, és ha ezek valamilyen hiba miatt (vagy mert egy rosszindulatú támadó átvette fölöttük a hatalmat) elkezdenek szabálytalanul működni akkor is csak saját memóriaterületükben, saját adatterületükben tehetnek kár, és csak a számukra kirendelt erőforrásokat abuzálhatják, hiszen minden mást csak a megbízható kernelhez intézett kérésekkel érhetnek el. (Amiket az ellenőriz. Ne feledjük, az IPC-t is a mikrokernel kontrollálja!) Összefoglalva: a mikrokerneles rendszer nagyobb stabilitása elsősorban abban rejlik hogy a hibás vagy rosszindulatú kód nagyon kis eséllyel fog kernel módban futni. Azonban van még ezenkívül is egy hasznos tulajdonságuk: (amivel azonban szintén nem biztos, hogy mindegyikük rendelkezik) a szerverek között futtathatunk egyet ami csak azzal foglalkozik, hogy folyamatosan ellenőrzi a többit hogy

képesek-e válaszolni, helyesen látják-e el a feladatukat, és ha valamelyik lefagyni, vagy „elromlani” látszik, akkor azt - szükség esetén – leállítja és újraindítja. Ez a szolgáltatás szintén nagyban növelheti egy operációs rendszer stabilitását, de ez a több-szerveres, mikrokerneles rendszerek rugalmasságával valósítható csak meg, egy monolitikus rendszerben hasonló megoldás elképzelhetetlen.

Ha az operációs rendszerek stabilitásáról és biztonságáról beszélünk még egy megoldást meg kell említenünk, még akkor is, ha napjainkban szinte sehol nem használják, ez pedig a nyelvi alapú védelem. Alapvetően más megközelítés, más filozófia ez, mint a fentiek. Arról van szó, hogy bizonyos programokat teljes lelki nyugalommal beengedünk a kernelspace-be, mivel egész egyszerűen *megbízunk* bennük, mégpedig azért mert egy olyan programozási nyelven íródtak melyben lehetetlen károsan működő programot írni. Az ötlet nem is újszerű igazából, hiszen ezt a módszert már a Burroughs B5000 számítógépeken is használták, ahol a megbízható nyelv szerepét az Algol töltötte be. (Bár ott ezt még inkább a szükség vezérelte, hiszen a gép nem rendelkezett MMU egységgel amivel a folyamatok külön címtere megvédhető lett volna.) Mindenesetre én nem emiatt a történelmi jelentőség (?) miatt foglalkozok a megoldással, hanem mert erre épül a Microsoft Research legújabbak fejlesztése, egy kísérleti operációs rendszer, mely egyelőre a Singularity névre hallgat. Ismerve a Microsoft üzletpolitikáját (nem szoktak pénzt ölni olyasmibe, amiből nem szándékoznak jóval többet viszontlátni,) és tudva hogy már idáig is komoly erőforrásokat öltek a rendszerbe, illetve hogy sikerült néhány a területükön igen nagy névnek számító kutató-fejlesztőt megnyerniük a projekthez, (a projektvezetők Galen Hunt és Jim Larus,) talán nem tűnik lehetetlennek a feltételezés hogy ezt a rendszert szánják (ha tényleg életképesnek bizonyul a koncepció) a történelmi (piaci) sikerű Windows termékcsalád utódjául.

A Singularity is mikrokerneles rendszer, de nagyban különbözik például a Minix-től: itt *minden* megbízhatónak ítélt folyamat kernel módban fut. Maga a mikrokernel, a szerverek és a felhasználó programok is egyetlen közös virtuális címtérben futnak. Hagyományos értelemben nem is beszélhetünk külön folyamatokról, hiszen semmiféle hardveres védelem nem választja el őket, hasonlóan egy hagyományos monolitikus kernel belsejéhez. Ennek megfelelően a Singularity fejlesztői nem is ezt a kifejezést használják, hanem SIP-nek (*Software Isolated Process*) nevezik ezeket a folyamatokat. És valóban: az elkülönítés teljesen szoftver alapú, a fordítóprogramra épül. A Singularity a mikrokernel egy kis részét kivéve –

ide tartozik például a HAL és a szemétgyűjtő, melyek Assembly és C++ nyelven íródtak – Sing#-ban íródott. A Sing# a Microsoft saját C# nyelvének továbbfejlesztése, kiegészítve azt egy formális előírásokon alapuló IPC mechanizmussal. A SIP-ek ezzel a szigorú típusosságra épülő IPC-vel, kétirányú csatornákon keresztül kommunikálnak egymással, a szerverekkel és magával a mikrokernellel is. (A Singularity esetében a kernelben implementált szolgáltatások a memóriafoglalás, a hardver-elérés ellenőrzése, valamint a szál-ütemezés és szinkronizáció.) Ezzel, és a ténnyel hogy az objektum orientált, és szigorúan biztonságos Sing# nyelven lehetetlen olyan programot lefordítani, amely hozzáférhetne más folyamatok adatszerkezeteihez is, egy a legortodoxabb klasszikus mikrokerneles rendszerekéhez fogható biztonság érhető el. (A Sing# elődje a C# sokak szerint a „Microsoft Java-ja" akarna lenni.) Mindeközben a teljesítmény sem szenved semmilyen hátrányt: mivel virtuálisan egyetlen címtérben fut minden nincsenek kontextus és módváltások, de még a szokásos megszakításkezeléssel sem kell számolnunk. Az egyetlen hátránya teljesítmény szempontból a Singularity modelljének, hogy egyáltalán nem támogatja egy folyamat dinamikus kibővítését végrehajtható kóddal (mint például egy plugin vagy betölthető kernel-modul) mivel ezzel veszélybe kerülne az adott folyamat megbízhatósága, így minden ilyen kis kiegészítő kód kénytelen saját SÍP-jében futni, és a szabványos IPC-n keresztül kommunikálni az anyafolyamattal. A teljesítmény érdekében egy ponton még fel is adja a Singularity a SIP-ek adatstruktúráinak tökéletes izolációját: lehetővé teszi egy osztott objektumverem használatát, mivel nagy mennyiségű adat mozgatása a folyamatok között a fent említett IPC-csatornákon keresztül nagyon lassú volna. A veremben továbbra is minden objektumnak csak pontosan egy SIP a tulajdonosa, de az megváltoztatható a normál IPC segítségével.

A Singularity-ben minden a rendszer részét képező programhoz tartozik egy leíró metaadat mező, amivel külső ellenőrző programok verifikálhatják őket mielőtt elindulnának. Az első verifikálást maga a fordító végzi, amikor az eredeti forrásból elkészíti a közbülső bájtkódot. Aztán ez a bájtkód további ellenőrzésen esik át külső verifikáló programok által, végül még a háttérben futó just-in-time fordítónak is lehetősége van azt leellenőrizni mielőtt valódi végrehajtható gépi kódot csinálna belőle. Ez a több lépéses ellenőrzés biztosítja, hogy a SIP-ekben valóban csak biztonságos kód futhat.

Az operációs rendszerek stabilitásának mérésére nem igazán létezik közkeletűen elfogadott metrika. Az egyes népszerű rendszerek hívei között internetes fórumokon évek, évtizedek óta megy a „vérre menő” vita ebben a kérdésben, és bár ebbe a laikus felhasználókon kívül gyakran komoly szakemberek is bekapcsolódnak (lásd a Torvalds-Tanenbaum vita,) mégsem sikerült még soha egyik ilyenben sem konszenzusra jutni. Éppen ezért nem is vállalkozom itt arra, hogy megpróbáljam egzakt módon meghatározni hogy mennyivel stabilabb egy mikrokerneles (vagy éppen egy virtualizált, vagy nyelvi védelemmel operáló) rendszer mint egy hagyományos monolitikus, illetve hogy megéri-e ez a különbség a hátrányt ami a teljesítményben jelentkezik. Az hogy egy mikrokerneles rendszer nagy általánosságban stabilabb az nem lehet vitás, nem csak a fenti elméleti okfejtés, de a gyakorlati tapasztalat is alátámasztja. De hogy kinek van szüksége erre a megnövekedett biztonságra? Ez attól függ, mire használjuk azt a bizonyos számítógépet, amin az operációs rendszer fut. A mai mindennapi tapasztalatok azt mutatják, hogy egy közönséges otthoni/irodai munkaállomásra, melyet csak szórakozásra, multimédiás célokra, munkavégzésre vagy internetezésre használnak bőven elegendő az a stabilitás is, amit egy hagyomány monolitikus rendszer, mint a Linux vagy a Windows tud nyújtani. Web- vagy LAN-szervereknél már egy kicsit komolyabbak az elvárások megbízhatóságok terén, de néhány speciális alkalmazási területet kivéve itt is a monolitikus rendszereket használják a gyakorlatban, főleg ahol nem okoz katasztrofális hatásokat egy-egy rövidebb kimaradás, vagy ahol a kiszolgáló feladatok több külön-külön operációs rendszer példányt futtató szervergép között vannak elosztva, ahol ha az egyiket újra is kell indítani addig a többi át tudja venni a feladatát. Egy alaposan tesztelt, és sok éve patch-elt monolitikus rendszer igen-igen megbízható tud lenni: egy átlagos Linux gép, ha hardveres akadálya nincs, hetekig vagy hónapokig képes leállás vagy komolyabb rendszerhiba nélkül futni, míg egy Windows rendszer átlagban nagyságrendileg azonos, de valamivel rosszabb adatokat produkál a fenti jellegű használat mellett. (Legalábbis ha a rendszerek munkaállomásokra optimalizált változatairól beszélünk, de ez az eltérés számos vélemény miatt betudható akár annak is, hogy az átlagos Windows felhasználó kevesebb tapasztalattal rendelkezik, mint egy átlagos „linuxos.”) Ilyen típusú rendszerekben a megbízhatóság nem kritikus, értem ez alatt, hogy nem látni értelmét „beáldozni” 10-20-50% átlagsebességet cserébe azért, hogy az előforduló rendszerleállást okozó hibák gyakorisága 10^{-3} %-osról 10^{-7}%-os nagyságrendűre csökkentsük.

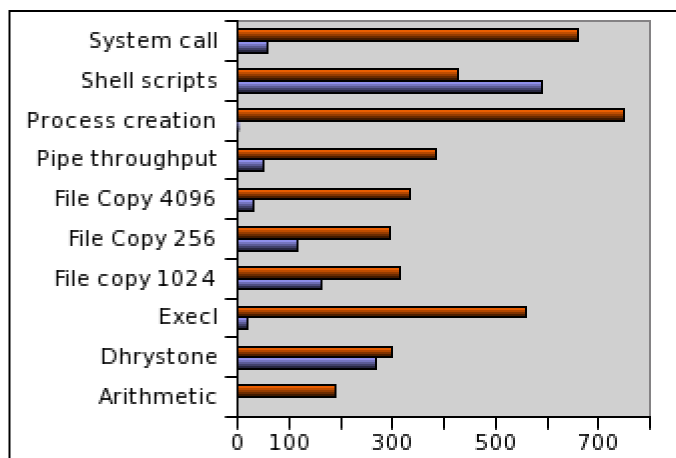
(Ezek az adatok itt csak nagyon durva becslések, a teljesítmény adatokat lásd majd lentebb.) Természetesen léteznek olyan felhasználási területek is ahol valóban kritikus a megbízhatóság és a biztonság, és nincs az a (hardverfejlesztéssel kompenzálható) sebességvesztés, ami ne érné meg még egy kis plusz biztonságért cserébe. Ilyenek lehetnek például (tényleg csak a teljesség igénye nélkül) az orvosi rendszerek, egy repülőgép irányítószoftvere, egy atomerőmű vezérlőszámítógépe stb. Létezik a mikrokerneles rendszereknek egy másik felhasználási területe is, ahol egyáltalán nem a nagyobb stabilitás a lényeg: ezek pedig a mobil- és beágyazott rendszerek. Itt a multiszerveres felépítés azon előnyös tulajdonsága van kihasználva, hogy az operációs rendszer legalapvetőbb szolgáltatásai is rendkívül dinamikusan kezelhetőek: mindig csak azokat a szolgáltatásokat kell elindítani (bluetoot vagy kamera driver stb.) amikre éppen szükség van, így takarékoskodni lehet a szűkös tárkapacitással.

(Megjegyzés: a fentebb olvasható csoportosítás az ábrákkal együtt Andrew S. Tanenbaum, Jorrit N. Herder, és Herbert Bos a computer.org-on megjelent Can We Make Operating Systems Reliable and Secure? című cikkéből származik.)

És most térjünk rá a másik kardinális kérdésre, a teljesítményre. Hagyományosan egy operációs rendszer sebessége (vagyis az a tényező, amivel az operációs rendszer befolyásolja a programok futási sebességét) leginkább a kontextusváltások függvénye, azaz azon múlik, hogy hányszor kell a processzornak végrehajtania az állapotrögzítés, módváltás, veremkezelés, állapot-visszaállítás költséges műveleteit. (Mivel most az egyes rendszermodellek közötti különbségről beszélek, eltekintek attól hogy természetesen a szolgáltatást nyújtó rutin kódja is lehet különböző hatékonyságú különböző operációs rendszerekben – jelen esetben feltételezem hogy ugyanaz a kód fut le mindkét esetben, a futási időben jelentkező különbséget csupán a kontextusváltások száma adja.) Egy multiszerveres mikrokerneles rendszerben a kontextusváltások száma értelemszerűen sokkal nagyobb, mint egy monolitikusban. Az utóbbiban egy hívás során egy (pontosabban egy pár) váltás történik: a hívó felhasználó folyamat elhelyezi a küldendő paramétereket a veremben, majd meghívja a megfelelő kernel rutint megszakítással vagy más módszerrel, ezután elmentődik az aktuális állapota, a kernel kezeli a hívást (úgy hogy közben saját magán belül nem történik semmilyen kontextusváltás, csakis eljáráshívások,) majd elhelyezi az esetleges visszatérési értékeket a veremben és visszakerül a vezérlés a hívó folyamathoz. Ezzel

szemben a mikrokerneles rendszerben egy szolgáltatás igénybevételéhez az IPC-n keresztül kell üzenetet küldeni a megfelelő szerverfolyamatnak, amit viszont a mikrokernel bevonásával lehet csak megtenni. Azaz legjobb esetben is kétszer két váltás történik: a felhasználó folyamatból a kernelbe, majd onnan a szerverfolyamatba, és vissza. Természetesen olyan eset is felmerülhet, hogy a szerver más szerverek szolgáltatásait kénytelen igénybe venni, ami további IPC-t és kontextusváltásokat jelent. Ebből látszik az is hogy mi az, amin igazán múlhat egy mikrokerneles, sokszerveres rendszer sebessége: mivel a kontextusváltások a rendszer szerkezetéből adódóan elkerülhetetlenek, a lényegi sebességoptimalizálás csak az IPC finomításával érhető el. (Persze azért a kontextusváltások száma és költsége is optimalizálható valamelyest, olyan módszerekkel, mint például amilyet az Apple által fejlesztett Vanguard mikrokernel használ: a több szerverhez tartó üzeneteket láncolásával. Mindenesetre a monolitikus rendszerekéhez hasonló nagyságrend nem érhető el.) A hagyományos monolitikus UNIX rendszerekben alkalmazott IPC (cső, illetve socket megoldások) messze nem voltak elég hatékonyak egy használható mikrokerneles rendszer felépítéséhez, így azokban kivétel nélkül valamilyen más mechanizmus van implementálva. Persze még így is óriási többletköltséget jelentett az IPC minden szolgáltatáshívás során. Az optimalizálására több fajta megoldás is született: a Mach kernel például a processzor MMU egységét használta a memórialapok tulajdonosainak cserélgetéséhez hogy ne kelljen nagy mennyiségű adatot másolgatni a felhasználó folyamatok címterei között. Azonban közmegegyezés szerint a leghatékonyabb IPC-vel (és így a legjobb teljesítménnyel) a jelenlegi mikrokernelek közül az L4 rendelkezik. (Tulajdonképpen ma már helyesebb kernel-családról beszélni, hiszen Jochen Liedtke eredeti fejlesztését azóta már sokan újraimplementálták.) Az L4 kernelt szokás második generációs mikrokernelnek is nevezni, mivel még az eredeti Mach kernelhez képest is jelentősen leegyszerűsített és lekicsinyített kernel-képpel (és optimalizált IPC-vel) dolgozik. Mikor Liedtke 97-ben megalkotta az első L4-et, méréseket készített annak eldöntésére, hogy milyen teljesítményre képes az egy hagyományos monolitikus rendszerhez, illetve egy első generációs mikrokernelhez képes. Ehhez Liedtke a szabad forráskódú Linux kernelt használta fel, azaz annak eredeti formáját hasonlította össze egy a Mach kernel felett futó változattal (ez az MkLinux) illetve egy L4-re épülő variációval. (Ez a rendszer is letölthető az internetről L4Linux néven.) Liedtke és kutatótársai arra a végkövetkeztetésre jutottak, hogy az alkalmazások szintjén a

teljesítményveszteség 5-10%-ra tehető az L4 kernel esetén a hagyományos monolitikus modellhez képest. Míg az természetesen távol álljon tőlem, hogy megkérdőjelezzem Liedtke pártatlanságát a kérdésben, de van néhány dolog, amiről nem szabad megfeledkeznünk: egyrészt az L4Linux nem egy hagyományos multiszerveres mikrokerneles rendszer, sőt, ahogy fentebb láttuk bizonyos szempontból tekinthető a virtualizált monolitikus rendszer egy változatának is. Másrészről szigorúan a felhasználó programok szemszögéből vizsgálva a kérdést kicsit csalóka általános következtetéseket levonni, mivel bár egy program futási idejének általában kisebb hányadát teszik ki a rendszerhívások kiszolgálásával kapcsolatos adminisztratív teendők (kontextusváltás, IPC) ez az arány programról programra változhat aszerint, hogy az mennyire gyakran hívja ezeket a rutinokat. Míg a téma egy másik nagy szakértője, Andrew S. Tanenbaum is 5-10% tette egy modern mikrokerneles rendszer teljesítménybeli veszteségét egy klasszikus monolitikus rendszerrel szemben, egy az lwn.net-en megjelent teszt (mely Tanenbaum saját „mintarendszerét" a Minix3-at hasonlította össze a Linux-szal) tanulsága szerint ez a különbség valójában a sokszorosa is lehet az említettnek, ha csupán a rendszerhívások hatékonyságát nézzük. (Ennek a tesztnek az eredményei és Liedtke mérései is megtalálhatóak a függelékben.)



A Minix3 (piros) és a Linux (kék) eredményei a Unixbench tesztprogramon futtatva.

2. e, Más modellek

Bár ennek a dolgozatnak a tárgya a monolitikus és a mikrokerneles rendszerek összehasonlítása, de úgy érzem nem lenne teljes a kép ha nem említenék meg néhány alternatív modellt is. Ezek per pillanat közel sem használtak olyan széles körben, mint a fenti kettő, de a téma szempontjából annál érdekesebbek. Az alábbi kategóriák tehát gyakran felmerülhetnek egy kernelmodellekről szóló tanulmányban/vitában, még ha sokszor csupán elméleti alternatívaként is.

1. Nanokernelek: lényegében egy másik szó a második generációs mikrokernelekre, mint az L4. A kernelspace-ben futó tényleges rendszermag nagyon kicsi, csupán néhány kilobájt, minden lehetséges funkció külön felhasználói módú folyamatokba van kiszervezve. A külön kifejezés kialakulására a magyarázat hogy készítői és hívei mindenképpen meg akarták különböztetni az új generációs mikrokerneleket a Mach féle korábbi modellektől, melyek valójában sok olyan funkciót is a kernelben tartottak melyekre nem lett volna feltétlenül szükség.
2. Exokernelek: eredetileg a MIT (Massachusetts Institute of Technology) Parallel and Distributed Operating Systems csoportja által kifejlesztett technológia, melyet azóta többen is lemásoltak. A koncepció lényege hogy a kernel nem kényszerít semmilyen absztrakciót a felhasználói programra, az bizonyos jogosultság-ellenőrzéseket és erőforrás-elosztást leszámítva úgy éri el a hardvert ahogy neki tetszik.
   A programok egészen konkrét erőforrásokra (abszolút memóriacímek, lemezblokkok stb.) és a kernel csak azt ellenőrzi, hogy az adott erőforrást pillanatnyilag nem foglalja-e valami más. Elsősorban olyan céllal jött létre a koncepció, hogy a hardver lehetőségeit a lehető legjobban kihasználni tudó programokat futtató operációs rendszer készülhessen. Általánosságban persze jóval körülményesebb ilyen operációs rendszerre fejleszteni.
3. Hibrid kernelek: rájuk természetesen nem vonatkozik a fenti megjegyzés a gyakorlati jelentőség hiányáról, hiszen valójában szinte minden ma széles körben operációs rendszer ebbe a kategóriába tartozik. Valamiféle átmenetet képeznek a tiszta monolitikus és mikrokerneles modell között. Általában létezik egy nem minden szolgáltatást tartalmazó kernel, és a kimaradó szolgáltatások külön folyamatban

futnak, de részben szintén kernel módban. Ide tartoznak például a Microsoft NT alapú Windows-ainak kerneljei is, amikben van egy viszonylag jól körülhatárolt „mikrokernel," de azon kívül még óriási mennyiségű kód fut a privilegizált módban, pl. a Unix alapú megoldásoktól eltérően még a grafikus felhasználói felület ablakkezelő alrendszere is.) Szintén hibrid kernellel rendelkezik az ugyancsak nagyon elterjedt Mac OSX (mely a Mach mikrokernelre épül, de számos szolgáltatást a BSD Unixokból átvett kernel módban futó kód lát el), és bizonyos szempontból még a hagyományos Linux is, hiszen támogat például userland fájlrendszer-meghajtókat.) Számos vélemény szerint a kifejezés eleve csupán reklámfogás, hiszen amúgy is nagyon kevés operációs rendszer van, ami tisztán mikrokerneles vagy monolitikus lenne az elméleti modell minden alaptulajdonságával.

3. Összefoglalás

Míg világunkat egyre inkább a számítógépek irányítják, és míg ezekre a számítógépekre egyre egyszerűbben, egyre gyorsabban és egyre praktikusabban írhatjuk a különböző automatizált fejlesztőeszközeink segítségével a programjainkat, addig hajlamosak vagyunk megfeledkezni róla hogy a ezeket a gépeket is (majdnem) mind operációs rendszerek vezérlik. Ha mint programozók szemléljük ezt a tényt kénytelenek vagyunk belátni hogy akármilyen alaposan teszteljük és vizsgáljuk át a programunkat, akármennyire gondosan is optimalizáljuk, az hogy milyen megbízhatóan és milyen sebességgel fut nagyban fog függeni az operációs rendszertől is. Ha mint felhasználókra tekintünk magunkra (és végső soron mind azok vagyunk,) akkor pedig főleg nem mindegy számunkra, hogy hogyan futnak nagy általánosságban a programjaink.

Mióta megjelentek a biztonságos operációs rendszerek írását lehetővé tevő védett módú processzorok, azóta folyik a vita a rendszertervezők között arról, hogy két rendszermodell közül melyet implementálni. A két modell közötti lényegi különbség hogy az operációs rendszer kódjának mekkora részét kívánják a processzor privilegizált üzemmódjában futtatni. Az egyik modell teljesítményt és könnyebb implementálhatóságot, a másik megnövelt stabilitást és rugalmasságot kínál. Igen hosszú ideje a monolitikus rendszermodell a domináns a gyakorlatban, (leszámítva néhány speciális alkalmazási területet) annak ellenére is hogy a '90-es évek elején sok szakértő már a kihalását jósolta. Ennek ellenére fennmaradt, sőt máig él és virul olyan rendszerek képében, mint a Windows vagy a legtöbb Unix-klón rendszer. Ennek okai sokfelé kereshetőek: egyrészt a gépekből kisajtolható teljesítmény minél jobban történő kihasználása a jóslatokkal szemben máig kritikus fontosságúnak ítélt tényező a felhasználók szemében, másrészt a mikrokerneles rendszer előnye nem igazán könnyen mérhető, számszerűsíthető tulajdonságok. Hozzájárulhatott ehhez a modell nehezebb implementálhatósága is, valamint a tény hogy egy ilyen rendszer mögött sem áll olyan gazdasági erejű szoftvercég, mint a Microsoft, vagy olyan fejlesztői közösség, mint a Linuxé.

Mindezek ellenére napjainkban a kérdés ismét előtérbe kerülni látszik a beágyazott és célrendszerek elterjedésével, melyeknél egyáltalán nem alapvető fontosságú a feldolgozási

sebesség, de a megbízhatóság és a rugalmasság annál inkább, és amelyek így ideális felhasználása környezetet jelentenek a mikrokerneles rendszereknek. Emellett mára a hardverek teljesítménye soha nem látott teljesítményt ér el (manapság látszik elkésve bár de beigazolódni a '90-es évek eleji jóslat miszerint minden területen, még a munkaállomásoknál is elterjednek a sokprocesszoros rendszerek,) így egyre inkább elfogadhatónak tűnik a (mikrokerneles rendszerek fejlesztése során amúgy is egyre kisebbre szorított) teljesítményveszteség. A személyi számítógépek átlagfelhasználójának képe is átalakulni látszik: manapság egyre többen csak kényszerből, a munkájukhoz használnak számítógépet, és amíg azt el tudják rajta végezni addig számukra fontosabb lesz a biztonság a sebességnél.

Mindezt végiggondolva érdemes lehet feltenni a kérdést: merre tovább? Mit hoz a jövő? Szerény véleményem szerint a már „kialakult" platformokon, mint a személyi gépek és a szerverek piacán a közeljövőben nem várható jelentős változás. Várhatóan továbbra is a Windows, illetve a különböző Unix-szerű operáció rendszerek fogják uralni ezt a területet. Ennek oka egyrészt az, hogy ezen gépek felhasználóinak/üzemeltetőinek egyszerűen nincs szükségük arra a plusz stabilitásra amit egy mikrokerneles rendszer nyújthat. másrészt ma ezen a területen csak egy igen kiforrott és jól támogatott rendszer tudná felvenni a versenyt a fent említett monolitikus óriásokkal, ilyen mikrokerneles operációs rendszer viszont pillanatnyilag egyszerűen nincs a láthatáron. (Se olyan, ami néhány éven belül meghatározóvá válhatna.) A megbízhatóság-kritikus rendszereken minden bizonnyal marad a mikrokerneles modell hegemóniája, mivel ezekhez egy monolitikus rendszer stabilitása továbbra sem lesz elég, a mobil rendszerek piaca pedig pillanatnyilag függőben van ebből a szempontból. Saját becslésem szerint itt is inkább a monolitikus operációs rendszerek térhódítása várható, mivel ezen készülékek tárkapacitása és teljesítménye egyre nő, (részben megszüntetve a fentebb tárgyalt kényszerhelyzetet) ugyanakkor folyamatosan egyre nagyobb sokoldalúságot, és egyre kiterjedtebb használhatóságot várnak el tőlük, aminek egy már jól bevált nagygépes monolitikus rendszer átirata talán jobban meg tud felelni mint egy kísérleti mikrokerneles megoldás.

Visszatérve a dolgozatom eredeti fókuszát jelentő munkaállomásokra és szerverekre szánt operációs rendszerek piacára, ezen a területen meggyőződésem szerint nem várható modellváltás egészen addig, amíg meg nem jelenik egy a jelenlegi piacvezetőkéhez hasonló fejlesztői és marketing háttérrel rendelkező rendszer. Ilyenné hosszútávon is csak a

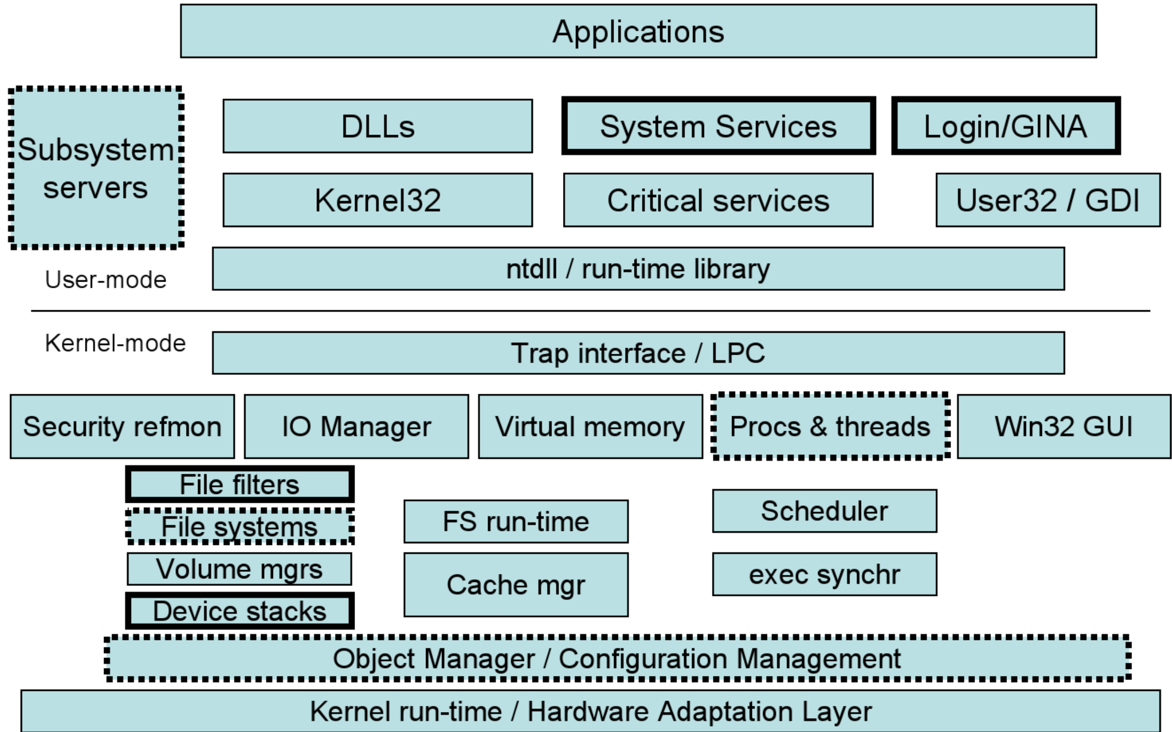
Singularity-nek van esélye válni, amennyiben a Microsoft kitart a projekt mellett, az L4 és Mach típusú kernelek pedig megmaradni látszanak kísérleti jellegű rendszereknek.

4. Irodalomjegyzék

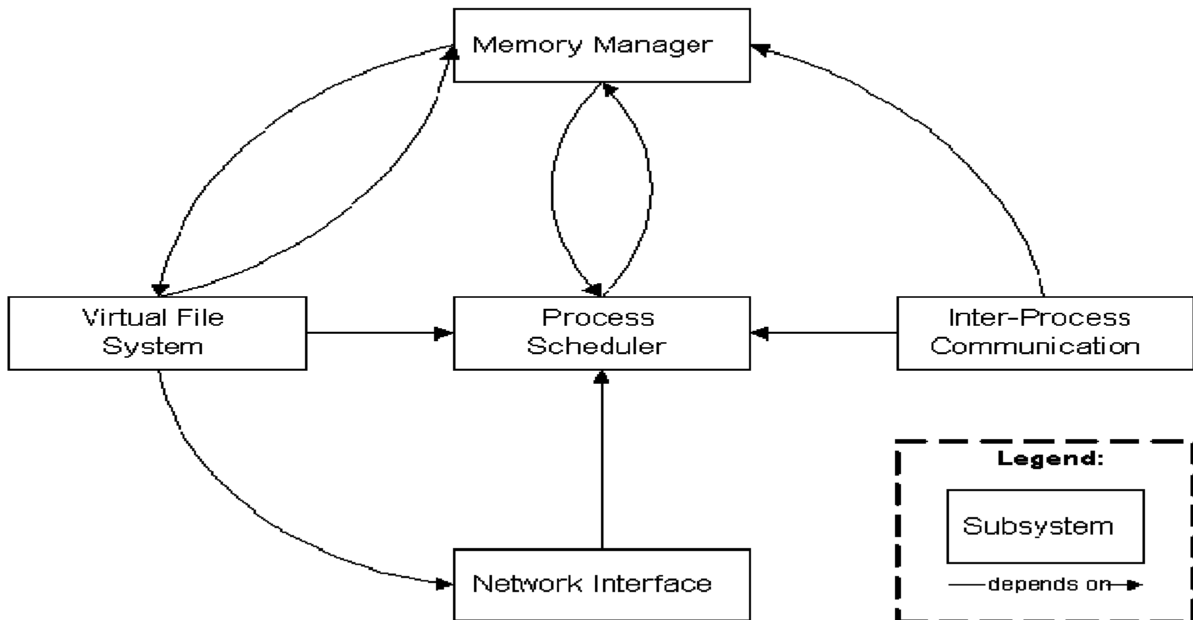
- Prof. Andrew S. Tanenbaum: Modern Operating Systems (2nd Edition)
- Daniel P. Bovet, Marco Cesati: Understanding the Linux Kernel (1st Edition)
- http://www.oreilly.com/catalog/opensources/book/appa.html (a Linus Torvalds - Andrew Tanenbaum vita)
- Andrew S. Tanenbaum, Jorrit N. Herder, Herbert Bos: Can We Make Operating Systems Reliable and Secure? (URL: http://www.computer.org/portal/site/computer/menuitem.5d61c1d591162e4b0ef1bd108bcd45f3/index.jsp?&pName=computer_level1_article&TheCat=1005&path=computer/homepage/0506&file=cover1.xml&xsl=article.xsl&)
- Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg: The performance of µ-kernel-based systems (URL: http://portal.acm.org/ft_gateway.cfm?id=266660&type=pdf&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618)
- Real World Technologies Forum, „Hybrid (micro)kernels" téma. (Több hozzászóló, köztük Linus Torvalds. URL: http://www.realworldtech.com/forums/index.cfm?action=detail&id=66595&threadid=66595&roomid=2)
- Mark Russinovich: Inside the Windows Vista Kernel (URL: http://www.microsoft.com/technet/technetmag/issues/2007/02/VistaKernel/)
- Galen C. Hunt, James R. Larus: Singularity Design Motivation (URL: ftp://ftp.research.microsoft.com/pub/tr/TR-2004-105.pdf)
- Sealing OS Processes to Improve Dependability and Security (URL: ftp://ftp.research.microsoft.com/pub/tr/TR-2006-51.pdf)
- An Overview of the Singularity Project (URL: ftp://ftp.research.microsoft.com/pub/tr/TR-2005-135.pdf)
- Úlfar Erlingsson, Athanasios Kyparlis: Microkernels (URL: http://www.cs.cornell.edu/Info/People/ulfar/ukernel/ukernel.html)
- J. Bradley Chen, Brian N. Bershad : The impact of operating system structure on memory system performance (URL: http://portal.acm.org/ft_gateway.cfm?id=168629&type=pdf&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618)
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne: THE MACH SYSTEM (URL: http://library.nocrew.org/lib/os/Mach.txt)
- Jochen Liedtke : Improving IPC by kernel design (URL: http://portal.acm.org/ft_gateway.cfm?id=168633&type=pdf&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618)
- Benjamin Roch: Monolithic kernel vs. Microkenrel (URL: http://www.vmars.tuwien.ac.at/courses/akti12/journal/04ss/article_04ss_Roch.pdf)
- Comparing Linux and Minix (lwn.net cikk) (URL: http://lwn.net/Articles/220255/)

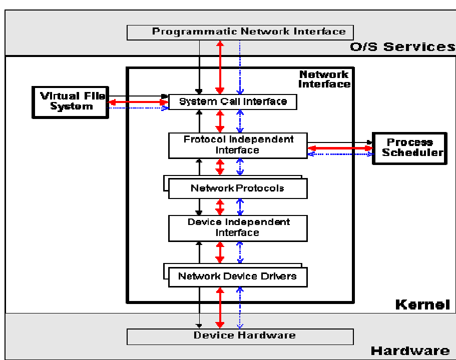
5. Függelék

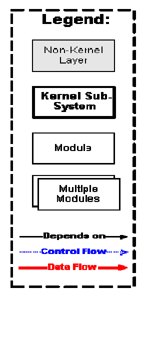
A Windows NT kernel felépítése:

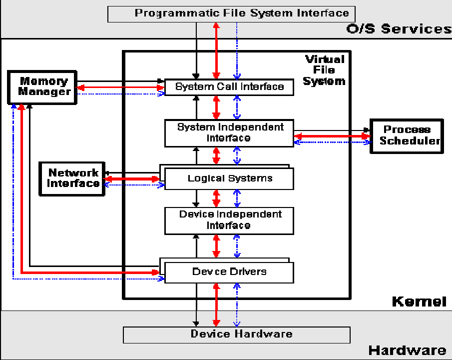


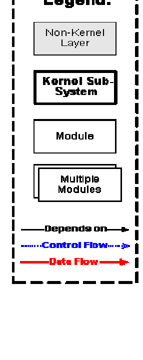
A Linux kernel felépítése:

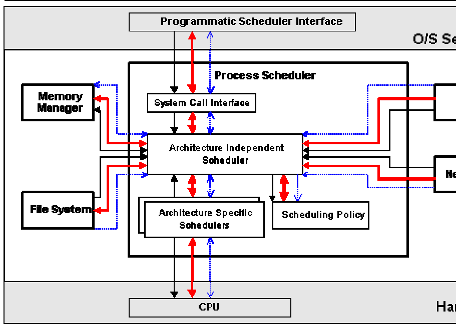


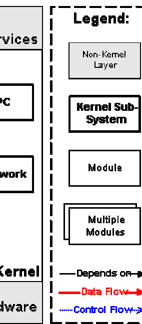


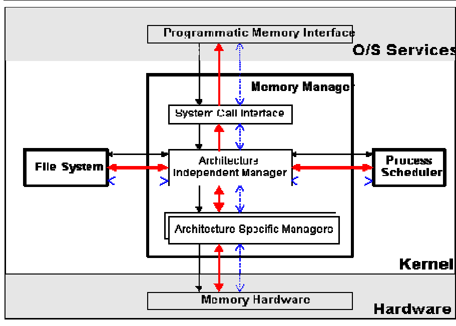


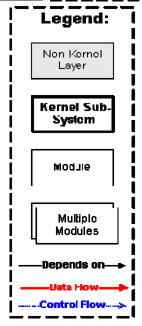












Liedtke eredményei az lmbench-el és a hbench-OS-el történő összehasonlításból:

Test	Linux	L4Linux libc.so	L4Linux trampoline	MkLinux in-kernel	MkLinux user
***lmbench* Results**					
Latency [μs]					
write to /dev/null	2.00 (0%)	5.26 (10%)	7.80 (6%)	24.33 (9%)	128.97 (2%)
Null Process	973 (1%)	2749 (4%)	2765 (1%)	3038 (1%)	3601 (1%)
Simple Process	7400 (1%)	12058 (2%)	12393 (1%)	14066 (1%)	19667 (1%)
/bin/sh Process	42412 (1%)	61115 (7%)	62353 (1%)	73201 (2%)	106853 (1%)
Mmap Latency	52.20 (2%)	64.28 (7%)	69.35 (8%)	345.33 (2%)	566.06 (1%)
2-proc ctxsw	7.00 (0%)	16.22 (6%)	18.20 (6%)	78.67 (9%)	79.87 (7%)
8-proc ctxsw	12.40 (4%)	22.22 (6%)	28.00 (4%)	85.67 (3%)	96.26 (6%)
Pipe	29.00 (2%)	52.07 (7%)	69.40 (6%)	308.33 (1%)	722.42 (2%)
UDP	159.40 (3%)	243.02 (4%)	263.80 (2%)	613.33 (4%)	1040.26 (2%)
RPC/UDP	321.40 (1%)	526.57 (3%)	528.80 (3%)	1095.33 (4%)	1743.29 (2%)
TCP	207.40 (2%)	287.57 (4%)	308.80 (5%)	562.00 (4%)	1047.03 (2%)
RPC/TCP	459.60 (2%)	729.76 (5%)	736.20 (4%)	1243.33 (4%)	2014.90 (2%)
Bandwidth [MB/s]					
Pipe	40.50 (2%)	37.61 (3%)	35.25 (3%)	13.11 (2%)	10.57 (2%)
TCP	18.03 (2%)	13.23 (2%)	13.41 (3%)	11.54 (1%)	10.88 (2%)
File reread	41.51 (1%)	40.43 (1%)	40.26 (3%)	37.51 (3%)	34.04 (2%)
Mmap reread	65.73 (1%)	54.96 (6%)	55.03 (7%)	61.54 (0%)	58.66 (7%)
***hbench:OS* Results**					
Latency [μs]					
getpid	1.69 (0%)	4.55 (1%)	6.91 (1%)	19.14 (1%)	111.9 (1%)
write to /dev/null	2.74 (0%)	6.67 (5%)	8.20 (4%)	26.30 (1%)	124.1 (1%)
Null Process	983 (1%)	2561 (1%)	2904 (1%)	3101 (1%)	3572 (1%)
Simple Process	7490 (1%)	12431 (1%)	12433 (1%)	14144 (1%)	19255 (0%)
/bin/sh Process	40864 (3%)	58845 (1%)	57968 (1%)	69990 (1%)	100763 (1%)
Mmap Latency 4KB	25.2 (0%)	35.0 (2%)	49.4 (2%)	242.7 (1%)	439.6 (1%)
Mmap Latency 8MB	53.7 (1%)	54.0 (2%)	74.9 (1%)	360.1 (1%)	561.9 (1%)
ctx 0K 2	8.05 (2%)	17.1 (4%)	20.0 (3%)	69.6 (3%)	79.9 (2%)
ctx2 0K 2	8.45 (3%)	17.0 (3%)	16.7 (6%)	76.2 (2%)	88.6 (3%)
Pipe	31.0 (2%)	62.3 (3%)	78.99 (3%)	316.1 (1%)	721.6 (1%)
UDP	154 (1%)	214 (1%)	251 (3%)	625 (1%)	1037 (1%)
RPC/UDP	328 (2%)	554 (2%)	577 (3%)	1174 (1%)	1763 (1%)
TCP	206 (2%)	264 (2%)	302 (1%)	568 (1%)	1030 (1%)
RPC/TCP	450 (2%)	754 (2%)	760 (3%)	1344 (1%)	2035 (1%)
Bandwidth [MB/s]					
Pipe 64KB	40.3 (1%)	35.5 (1%)	32.6 (2%)	12.7 (1%)	10.4 (2%)
TCP 64KB	18.8 (1%)	14.6 (1%)	14.1 (1%)	11.6 (1%)	9.4 (2%)
File read 64/64	35.3 (1%)	34.5 (4%)	32.2 (1%)	32.7 (3%)	30.1 (4%)
Mmap reread 64KB	97.5 (1%)	91.4 (1%)	78.8 (1%)	89.4 (1%)	77.7 (3%)

A unixbanch kimenete minix3-on futtatva:

```
BYTE UNIX Benchmarks (Version 4.0.1)
  System --
  Start Benchmark Run: Fri Feb  2 18:14:23 GMT 2007
   1 interactive users.
   18:14  up 0:18,  1 user, load averages: 0.31, 0.06, 0.86
  -rwxr-xr-x 1 bin  operator  74752 May  3  2006 /bin/sh
  /bin/sh: MINIX-PC 32-bit executable, sep I&D stripped
  /dev/c0d0p0s0       16384      11232       5152  32%     6%   /
Dhrystone 2 using register variables     3092311.2 lps   (10 secs, 10 samples)
Arithmetic Test (type = double)           5094.5 lps   (10 secs, 10 samples)
System Call Overhead                     93625.3 lps   (10 secs, 10 samples)
Pipe Throughput                          63605.0 lps   (10 secs, 10 samples)
Pipe-based Context Switching             25416.0 lps   (10 secs, 10 samples)
Process Creation                            66.2 lps   (30 secs, 3 samples)
Execl Throughput                            89.0 lps   (29 secs, 3 samples)
File Read 1024 bufsize 2000 maxblocks    129238.0 KBps  (30 secs, 3 samples)
File Write 1024 bufsize 2000 maxblocks   129021.0 KBps  (30 secs, 3 samples)
File Copy 1024 bufsize 2000 maxblocks     63963.0 KBps  (30 secs, 3 samples)
File Read 256 bufsize 500 maxblocks       39372.0 KBps  (30 secs, 3 samples)
File Write 256 bufsize 500 maxblocks      40483.0 KBps  (30 secs, 3 samples)
File Copy 256 bufsize 500 maxblocks       19652.0 KBps  (30 secs, 3 samples)
File I/O                                 no measured results
Shell Scripts (1 concurrent)              1718.6 lpm   (60 secs, 3 samples)
Shell Scripts (8 concurrent)               356.0 lpm   (60 secs, 3 samples)
Shell Scripts (16 concurrent)              187.0 lpm   (60 secs, 3 samples)
Arithmetic Test (type = short)          291399.2 lps   (10 secs, 3 samples)
Arithmetic Test (type = int)            291317.8 lps   (10 secs, 3 samples)
Arithmetic Test (type = long)           291274.0 lps   (10 secs, 3 samples)
Arithmetic Test (type = float)           10712.8 lps   (10 secs, 3 samples)
Arithoh                                6632757.3 lps   (10 secs, 3 samples)
C Compiler Throughput                     3930.0 lpm   (60 secs, 3 samples)
Dc: sqrt(2) to 99 decimal places          3823.3 lpm   (30 secs, 3 samples)
Recursion Test--Tower of Hanoi           47784.5 lps   (20 secs, 3 samples)
```

INDEX VALUES

TEST	BASELINE	RESULT	INDEX
Arithmetic Test (type = double)	29820.0	5095.6	1.7
Dhrystone 2 using register variables	116700.0	3104553.2	266.0
Execl Throughput	43.0	90.1	21.0
File Copy 1024 bufsize 2000 maxblocks	3960.0	64322.0	162.4
File Copy 256 bufsize 500 maxblocks	1655.0	19676.0	118.9
File Copy 4096 bufsize 8000 maxblocks	5800.0	17197.0	29.6
Pipe Throughput	12440.0	62242.3	50.0
Pipe-based Context Switching	4000.0	25039.1	62.6
Process Creation	126.0	66.7	5.3
Shell Scripts (8 concurrent)	6.0	353.0	588.3
System Call Overhead	15000.0	90252.8	60.2
		=========	
FINAL SCORE		48.1	

A unixbanch kimenete hagyományos Linux-on futtatva:

```
 BYTE UNIX Benchmarks (Version 4.0.1)
  System -- Linux victim 2.6.17-2-486 #1 Wed Sep 13 15:56:30 UTC 2006 i686 GNU/Linux
  Start Benchmark Run: Sun Dec 24 02:58:01 MST 2006
   3 interactive users.
   02:58:01 up  1:26,  3 users,  load average: 0.46, 0.52, 0.63
  lrwxrwxrwx 1 root root 4 2006-12-24 01:14 /bin/sh -> bash
  /bin/sh: symbolic link to `bash'
  /dev/hda1           15124868   2596292  11760272  19% /
Dhrystone 2 using register variables     3488372.6 lps   (10 secs, 10 samples)
Arithmetic Test (type = double)          571325.7 lps   (10 secs, 10 samples)
System Call Overhead                     991727.9 lps   (10 secs, 10 samples)
Pipe Throughput                          480018.6 lps   (10 secs, 10 samples)
Pipe-based Context Switching             173718.5 lps   (10 secs, 10 samples)
Process Creation                           9409.1 lps   (30 secs, 3 samples)
Execl Throughput                           2399.4 lps   (29 secs, 3 samples)
File Read 1024 bufsize 2000 maxblocks    364678.0 KBps  (30 secs, 3 samples)
File Write 1024 bufsize 2000 maxblocks   194217.0 KBps  (30 secs, 3 samples)
File Copy 1024 bufsize 2000 maxblocks    124050.0 KBps  (30 secs, 3 samples)
```

File Read 256 bufsize 500 maxblocks 176081.0 KBps (30 secs, 3 samples)

File Write 256 bufsize 500 maxblocks 68001.0 KBps (30 secs, 3 samples)

File Copy 256 bufsize 500 maxblocks 48531.0 KBps (30 secs, 3 samples)

File Read 4096 bufsize 8000 maxblocks 514724.0 KBps (30 secs, 3 samples)

File Write 4096 bufsize 8000 maxblocks 334735.0 KBps (30 secs, 3 samples)

File Copy 4096 bufsize 8000 maxblocks 193880.0 KBps (30 secs, 3 samples)

Shell Scripts (1 concurrent) 1139.5 lpm (60 secs, 3 samples)

Shell Scripts (8 concurrent) 257.0 lpm (60 secs, 3 samples)

Shell Scripts (16 concurrent) 141.3 lpm (60 secs, 3 samples)

Arithmetic Test (type = short) 283289.6 lps (10 secs, 3 samples)

Arithmetic Test (type = int) 289010.9 lps (10 secs, 3 samples)

Arithmetic Test (type = long) 289037.6 lps (10 secs, 3 samples)

Arithmetic Test (type = float) 579447.8 lps (10 secs, 3 samples)

Arithoh nan lps (10 secs, 3 samples)

C Compiler Throughput 567.3 lpm (60 secs, 3 samples)

Dc: sqrt(2) to 99 decimal places 65798.6 lpm (30 secs, 3 samples)

Recursion Test--Tower of Hanoi 75585.1 lps (20 secs, 3 samples)

INDEX VALUES

TEST	BASELINE	RESULT	INDEX
Arithmetic Test (type = double)	29820.0	571325.7	191.6
Dhrystone 2 using register variables	116700.0	3488372.6	298.9
Execl Throughput	43.0	2399.4	558.0
File Copy 1024 bufsize 2000 maxblocks	3960.0	124050.0	313.3
File Copy 256 bufsize 500 maxblocks	1655.0	48531.0	293.2
File Copy 4096 bufsize 8000 maxblocks	5800.0	193880.0	334.3
Pipe Throughput	12440.0	480018.6	385.9
Process Creation	126.0	9409.1	746.8
Shell Scripts (8 concurrent)	6.0	257.0	428.3
System Call Overhead	15000.0	991727.9	661.2
			=========
FINAL SCORE			389.4

(Mindkét teszt ugyanazon a gépen futott: egy AMD Athlon 1700-as processzoron, egy ATA winchesteren elhelyezkedő 4 GB-os partíciókon lévő operációs rendszer példányokra.)

6. Köszönetnyilvánítás

Szeretnék köszönetet mondani témavezetőmnek, Dr. Fazekas Gábornak támogatásáért és szakmai iránymutatásáért.