

TDK - dolgozat

Pető Albert

Intézmény: Miskolci Egyetem

Kar: Gépészmérnöki és Informatikai Kar

Tanszék: Általános Informatikai Tanszék

Szerző: Pető Albert, BSc II. évfolyam

Témavezető: Vincze Dávid, egyetemi tanársegéd

Dolgozat címe:

Gyors rendszerállapot kinyerési szolgáltatás Linux operációs rendszeren



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Gyors rendszerállapot kinyerési szolgáltatás Linux operációs rendszeren

Pető Albert

Programtervező informatikus BSc, II. évfolyam

Konzulens:

Vincze Dávid

egyetemi tanársegéd

Általános Informatikai Tanszék

MISKOLC, 2012

Tartalomjegyzék

1. Bevezetés	1
2. Rendszerállapot lekérdezésről általában	2
2.1. Rövid ismertetés a rendszerhívásokról	2
2.2. A proc fájlrendszer	3
2.3. Monitorozás egy terhelt rendszeren	4
2.4. Alternatívák	6
3. A bővítéshez szükséges előismeretek	8
3.1. Modulok	8
3.2. A szolgáltatás típusa	9
3.3. Az adatátvitel típusa	9
3.4. Szűrés	11
3.5. Biztonság	11
3.6. Az adatok összegyűjtése	12
3.7. Szükséges adatok	13
4. Az új szolgáltatás	15
4.1. Alapvető tulajdonságok	15
4.2. Adatstruktúrák	17
4.3. Az implementáció részletei	19
4.4. Példaprogram	19
5. Teszteredmények	21
5.1. Használt eszközök	21
5.2. Vizsgált programok	22
5.3. Eredmények	22
5.3.1. Processzek számának növelése	22
5.3.2. Fokozott processzor terhelés esete	23
5.3.3. Kevert környezet	24
6. Összefoglalás	25
Irodalomjegyzék	26

1. fejezet

Bevezetés

A Linux operációs rendszer manapság az egyik legelterjedtebb operációs rendszer. Szabad felhasználhatósága és nyílt forráskódja mellett közkedvelt még stabilitása és testreszabhatósága miatt. Bár immár több mint két évtizedes pályafutása során egy jól kiforrott rendszerré vált, a felhasználók egyre növekvő elvárásainak megfelelni csak a folyamatos változás révén képes.

A Linux operációs rendszerre általában, mint teljes értékű termékre hivatkoznak (Linux disztribúció), ám valójában maga az operációs rendszer alapvetően a rendszermagban (Linux kernel) testesül meg. Egy rendszerszintű új funkció hozzáadása szinte mindig a kernelhez való új funkció hozzáadását jelenti. A fejlesztések intenzívek és sokrétűek. Többek között állandó téma a hatékonyságnövelés, teljesítményoptimalizálás. A sebesség növelését örök problémának tekinthetjük, hiszen a hardver és szoftver képességeivel együtt az igények is folyamatosan növekszenek.

E dolgozat fő célja is a hatékonyság növelése. Egy olyan új szolgáltatást mutat be a Linux kernelben, aminek a használatával a rendszer adminisztrátorai és felhasználói a már meglévő módszerekhez viszonyítva jelentősen kisebb erőforrásigénnyel képesek információt kinyerni a rendszer bizonyos állapotairól. Erre a célra jelenleg egy univerzális, megbízható, összességében jól bevált technika használható. Ám a rendszer terhelt, esetleges túlterhelt, kritikus időszakaiban – éppen akkor, amikor igazán szükség lenne az erőforrások használatának feltérképezésére – nem elég hatékony. Továbbá, bizonyos esetekben a kiszolgálókon gyakorta alkalmazott, periodikusan lefutó rendszerstatisztika készítő, rendszermegfigyelő alkalmazások erőforrásigényét is jelentősen növeli. Ezek a lehetséges helyzetek támasztják alá a bemutatott új módszer létjogosultságát, melynek gyorsasága ilyen értelemben kap igazán jelentős szerepet. A dolgozat a rendszermag bővítésén túl példa alkalmazásokat is bemutat, amelyek az új interfészt használják, továbbá a hagyományos és az új módszert összehasonlító mérések eredményeit is közli.

A bemutatott kutató munka a TÁMOP-4.2.1. B-10/2/KONV-2010-0001 jelű projekt részeként az Európai Unió támogatásával, az Európai Szociális Alap finanszírozásával valósul meg.

2. fejezet

Rendszerállapot lekérdezésről általában

Minden operációs rendszeren adódhatnak szituációk, melyek alkalmával szükséges a rendszer állapotainak vizsgálata. Típusos példa erre, amikor egy valamilyen szempontból túlterhelt, vagy hibásan működő rendszeren a hiba okait fel kell tárni. Komolyabb, egy időben akár több tucat felhasználó által használt rendszereken, illetve szervergépeken a rendszerállapotok lekérdezése már nem csak alkalmi, hanem egy ismétlődő feladat. Például statisztikák készítéséhez célszerű a rendszerállapot lekérdező programokat periodikusan futtatni.

Linux rendszereken az ilyen jellegű lekérdezésekre számos program létezik már. Használatuk már több tíz évre nyúlik vissza, az idő során ezek mind nagyon jól használható eszközökké váltak, sőt, kategóriájukban szinte már egyeduralkodóvá váltak. Főleg a rendszeradminisztrátorok eszközkészletének fontos elemei. Ezek közül néhány :

ps : ez a legáltalánosabb, leggyakrabban használt monitorozó program. Alapvetően a rendszeren futó processzek¹ pillanatnyi állapotáról szolgáltat információt. Az adatok szűrése könnyen elvégezhető vele.

top : a rendszer azon processzeinek adatait adja vissza, melyek valamilyen erőforrás használatát tekintve a legnagyobb igényűek. Ennek a programnak viszonylag sok időre van szüksége, hogy meg tudja állapítani, hogy egy adott időintervallumban mely processzek használták a legtöbbet az erőforrásokat. Általában a processzor a legfontosabb erőforrás.

fuser : adott fájlhoz megkeresi az őket használó processzeket

lsuf : alapvetően a rendszer összes használatban lévő fájlját listázza ki.

netstat : a hálózati információk kezelésére szolgáló elsődleges eszköz.

Ezen eszközök stabilitása és egységessége ugyanazon alapszik : a proc fájlrendszeren.

2.1. Rövid ismertetés a rendszerhívásokról

Mielőtt bemutatom a proc fájlrendszer működését, egy rövid bevezetést szeretnék írni a rendszerhívásokról.

¹Folyamat – vagy angolosan processz – alatt egy program futásban lévő példányát értjük

Mivel a mai modern operációs rendszereken egy időben számos – akár több ezer – folyamat is működhet egyszerre, valamilyen központosított vezérlésre van szükség, amely biztosítja, hogy ezek egymás működését nem befolyásolják rossz értelemben. Másrészt pedig a kényelmes programozás érdekében szükség van arra is, hogy a programozóknak ne kelljen a bonyolult hardver szintű funkciókkal foglalkozniuk, azaz, hogy a programok egy hardverfüggetlen, absztrakt környezetben futhassanak. Ezt a két feladatot tölti be az operációs rendszerek világában a kernel². Bár többféle kernel létezik [1], e dolgozat szempontjából elég csak a Linux kernel típusát jellemezni. Linux operációs rendszereken a kernel egy végrehajtható fájl, amely a bootolás után először fog végrehajtódni, így minden más folyamat keletkezését és működését kontrollálni tudja.

Tulajdonképpen egy folyamatnak a saját memóriájának és a regiszterek használatán kívül minden más művelethez a kernel segítségét kell kérnie, így valósul meg a biztonság és az absztrakció. Például a kernel segítségét kell kérnie más folyamatokkal való kölcsönhatáshoz, fájlok olvasásához, de még saját maga felfüggesztéséhez is. A kernellel való kölcsönhatás alapvető eszköze pedig a rendszerhívás. Ugyanis egy felhasználói processz nem érheti el közvetlenül a kernel memóriaterületét.

Egy rendszerhívás egy olyan speciális eljárás a kernelben, amely egy folyamatból speciális eszközökkel érhető el. Linuxon a rendszerhívások a programok számára minden szükséges funkciót implementálnak. Összesen kevesebb mint 400 db rendszerhívás van, bár számuk még bővíthet a jövőben. A dolgozat hátralévő részében felvetett problémák megértése szempontjából a fájlokkal kapcsolatos rendszerhívások megértése fontos lesz, ezért röviden ezeket is ismertetem.

Amikor egy folyamat egy fájl tartalmát olvasni szeretné, először ”meg kell nyitnia” a fájlt az `open()` rendszerhívással. Miután megnyitotta, különböző rendszerhívásokat használhat a fájl tartalmának kezeléséhez. A két leggyakoribb ilyen rendszerhívás a `read()` és `write()`. Használat után pedig ”be kell zárnia” a fájlt a `close()` rendszerhívással.

A hagyomány szerint a folyamatok megszakításokkal érhetik el a rendszerhívást. A megszakítás egy processzor instrukció, mely egy különleges állapot jelenlétét jelzi a kernelnek. A kernel egy megszakítás bekövetkezésekor felfüggeszti az aktuális instrukciófolyamot, és egy speciális kezelőrutinnak adja át a vezérlést. Bár már léteznek ennél gyorsabb módszerek is a rendszerhívások használatára³, a rendszerhívások köztudottan sok idővel járnak az általános eljárás-hívásokkal szemben.

2.2. A proc fájlrendszer

Linuxon a rendszer dinamikus állapotát jellemző összes információhoz egy szabványos módon lehet hozzájutni : a proc fájlrendszeren keresztül.[3] Ez a fájlrendszer virtuális, ami azt jelenti, hogy – a fájlrendszerek általános használatával ellentétben – nem fizikai tárolón létezik, hanem csak a memóriában. Alapelve, hogy a rendszer különböző részeinek állapotai számára különböző fájlokat hoz létre ezen a fájlrendszeren, amiket minden processz a szabványos fájlkezelő rendszerhívásokkal tud kezelni – legtöbbször csak olvasni.

²A kernel magyar megfelelője a rendszermag.

³Intel processzorokon a `sysenter` és `sysexit` processzor instrukciók kifejezetten rendszerhívások használatához lettek kifejlesztve.[2]

Amikor egy processz olvasni szeretne egy ilyen fájlból, a `read()` rendszerhívás a feladatot a `proc` fájlrendszerhez tartozó olvasó rutinak továbbítja.⁴ Ez a rutin pedig az olvasandó fájtól függően egy olyan eljárást hív meg, amely a hívó processz pufférébe a megfelelő adatokat teszi. Ami igazán megkülönbözteti ezt a többi fájlrendszertől, hogy az előbb említett eljárás nem egy tároló eszköztől, hanem közvetlenül a rendszer memóriájából gyűjti össze az adatokat.

A `proc` féle módszer talán a lehető legnagyobb, legátfogóbb szinten biztosítja az egységességet. A korábban felsorolt monitorozó programoknak nem kell különleges rendszerhívásokat ismerniük, és – mivel a fájlrendszerek külső megjelenése ritkán változik – a kernel komolyabb változásai sem befolyásolják a működésüket. Az adatok pedig a fájlrendszerek hierarchikus struktúráját örökölve jól áttekinthetően vannak elhelyezve. Példaként néhány fájl a Linux `proc` fájlrendszerből, mely a szokás szerint a `/proc` jegyzék alá van csatolva :

`/proc/meminfo` : átfogó információk a rendszer fizikai memóriájáról.

`/proc/filesystems` : a rendszer által használt fájlrendszerek listája.

`/proc/cpuinfo` : részletes információk a rendszer által jelenleg használt processzorokról.

`/proc/pid/` : a `pid` azonosítójú⁵ folyamathoz tartozó információk.

`/proc/pid/stat` : átfogó információk a `pid` azonosítójú processzről. A `ps` is használja ezt.

`/proc/pid/fd/` : a `pid` azonosítójú folyamat által megnyitott fájlokhoz tartozó fájl leíró⁶ illetve fájlnev.

2.3. Monitorozás egy terhelt rendszeren

A rendszerállapotok lekérdezésére idővel és erőforráshasználattal jár. Általában hétköznapi felhasználók ritkán használják a fent említett monitorozó eszközöket, és még ritkábban tapasztalnak nagy eltérést a program indítása és befejezése között. Rendszergazdák viszont gyakorta találkozhatnak azzal a problémával, hogy a monitorozó programok túl sokáig futnak és túl sok erőforrást vesznek igénybe. Talán a hétköznapi felhasználónak ismerősebb az az eset, amikor a rendszer használhatatlanul lassúvá válik, és az ennek okát felderíteni hivatott monitorozó programok a feladatuk elvégzése helyett szintén válaszképtelenné válnak.

A túl hosszú futási idő nem feltétlenül jelent gondot a monitorozás automatikus végrehajtása esetén, viszont növekedett erőforrásigénnyel jár, ami nem elhanyagolható mértékben növeli a rendszer terheltségét. Általában elmondható, hogy a rendszerállapot lekérdező programok futásideje nagy, és egy terhelt rendszer esetén még nagyobb.

Ennek oka pedig a következő : A monitorozó programok a `proc` fájlrendszeren tárolt fájlok átolvasásával működnek. Ez viszont sok fájl olvasásával jár, főleg ha a processzekről

⁴A fájlokkal kapcsolatos műveleteket mindig annak a fájlrendszernek a kezelőrutinjai fogják végezni, amelyen az adott fájl van.

⁵A kernel minden folyamathoz létrehozáskor egy természetes számot, a processz azonosítót – process id – rendel.

⁶Minden fájlhoz megnyitáskor a kernel egy fájl leíró – process descriptor – rendel.

egyenként is adatokat kell gyűjteni. Sok fájl olvasása pedig sok rendszerhívással jár, az pedig köztudott, hogy a rendszerhívás nem egy olcsó művelet. Sőt, ha egy program már olvas egy fájlt, akkor azt meg is nyitja, utána pedig be is zárja. Így minden egyes fájl olvasása legalább 3 – de mint látni fogjuk, általában több – rendszerhívással jár.

A következő vizsgálatot 48 processz esetén végeztem az `strace`⁷ programmal, a rendszer különösebb terhelése nélkül.

```
strace -c ps -ef
```

% time	seconds	usecs/call	calls	errors	syscall
92.94	0.001646	34	48		write
2.65	0.000047	0	160		read
1.86	0.000033	0	150		stat64
1.58	0.000028	0	164		close
0.96	0.000017	0	163		open
0.00	0.000000	0	1		execve
0.00	0.000000	0	2		time
0.00	0.000000	0	2		lseek
0.00	0.000000	0	1	1	access
0.00	0.000000	0	3		brk
0.00	0.000000	0	3		ioctl
0.00	0.000000	0	9	9	readlink
0.00	0.000000	0	8		munmap
0.00	0.000000	0	9		mprotect
0.00	0.000000	0	3		_llseek
0.00	0.000000	0	2		getdents
0.00	0.000000	0	23		rt_sigaction
0.00	0.000000	0	29		mmap2
0.00	0.000000	0	18		fstat64
0.00	0.000000	0	1		geteuid32
0.00	0.000000	0	2		fcntl64
0.00	0.000000	0	1		set_thread_area
0.00	0.000000	0	1		openat
0.00	0.000000	0	2		socket
0.00	0.000000	0	2	2	connect
100.00	0.001771		807	12	total

Láthatjuk, hogy `ps` által használt rendszerhívások túlnyomó többségét az `open()`, `close()`, `read()` és `stat64()`⁸ rendszerhívások adták.

A `fuser` vizsgálatával hasonló eredményekre juthatunk.

⁷Az `strace` program más program által végzett rendszerhívásokat figyel.

⁸A `stat64` átfogó információkat gyűjt egy fájlról.

strace -c fuser TDK.tex					
% time	seconds	usecs/call	calls	errors	syscall

30.46	0.000060	0	202	34	readlink
25.89	0.000051	1	93		read
15.23	0.000030	0	102		close
9.64	0.000019	0	61	7	open
9.64	0.000019	0	47		openat
9.14	0.000018	0	52		munmap
0.00	0.000000	0	1		write
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		getpid
0.00	0.000000	0	1	1	access
0.00	0.000000	0	3		brk
0.00	0.000000	0	3		mprotect
0.00	0.000000	0	1		getcwd
0.00	0.000000	0	58		mmap2
0.00	0.000000	0	85	2	stat64
0.00	0.000000	0	55		fstat64
0.00	0.000000	0	94		getdents64
0.00	0.000000	0	1		fcntl64
0.00	0.000000	0	1		set_thread_area
0.00	0.000000	0	1		socket

100.00	0.000197		863	44	total

Megállapíthatjuk tehát, hogy a `proc` fájlrendszert használó programok által használt rendszerhívások száma általában a processzek számának többszöröse, és a legtöbb közülük fájlkezeléssel kapcsolatos. Az, hogy általában ez a többlet elhanyagolható, azért van, mert általában néhány száz fájl olvasása még gyorsan lezajlódhat.

A rendszer egészére vonatkozó információkat leíró fájllok száma sosem változik, ezekből csak néhány tucat létezik. A processzekre egyenként vonatkozó információkat leíró fájllok száma viszont a processzek számától függ, azaz több ezer processz esetén több ezer bejegyzés lesz a `/proc` jegyzében, és mindegyik tartalmazni fogja a processzek tulajdonságait leíró fájllokat. Ilyen sok fájl egyenként való átvizsgálása már sokkal időigényesebb feladat, ami a monitorozó programok működésén is látszódik. Igazából ez a tény az, amely engem egy új rendszerállapot kinyerési interfész kidolgozására ösztönzött. Később részletesebben is bemutatom, hogy bizonyos körülmények között hogyan futnak le a `proc` fájlrendszert használó programok – az eredményeket az általam készített alternatív módszerrel összehasonlítva.

2.4. Alternatívák

Eddig még a vele járó sok rendszerhívás ellenére sem született a `proc` fájlrendszert teljes egészében helyettesítő alternatíva. Nem született, mert a `proc`, mint említettem, egy nagyon

jól használható fájlrendszer, és talán az összes lehetőség közül a hátrányai ellenére még mindig ez a legpraktikusabb. Egy ilyen módszert természetesen nem lehet egészében elvetni. A munkám során nem lecserélni igyekeztem ezt a rendszert, hanem kiegészíteni ott, ahol nem elég hatékonyan működik : a fájlok egyenként való végigolvasásában nagy fájlmenyiség – azaz sok folyamat – esetén.

3. fejezet

A bővítéshez szükséges előismeretek

Mielőtt bemutatom az általam kidolgozott alternatív lekérdezési módszert, szeretném az olvasónak a témához és az implementációhoz kapcsolódó előismereteket bemutatni. Továbbá azokat a problémákat és ötleteket is ismertetni szeretném, melyekkel az implementáció során találkoztam. Fontos tudni, hogy a kernel fejlesztése a C programozási nyelven történik.

Az előző fejezetben megállapított problémából kiindulva tehát a következő a cél: Linux operációs rendszeren egy olyan rendszerállapot kinyerési szolgáltatást létrehozni, melynek használata alacsonyabb futási időt eredményez, mint a proc fájlrendszer használata.

Egyértelmű, hogy ezt csak a kernel bővítésével lehet megvalósítani, hiszen egy felhasználói processz¹ a kernel közreműködése nélkül a közelébe sem férhet a rendszerinformációknak. A kernel pedig egyelőre csak a proc fájlrendszert biztosítja erre.

3.1. Kernel modulok

A Linux kernel monolitikus kernel, azaz egyetlen végrehajtható fájlban testesül meg. Mégis, egy tágabb értelemben két részből áll: az előbb említett végrehajtható fájlból, és kernel modulokból. Egy kernel modul valamilyen végrehajtható kódot tartalmazó, újralinkelhető formátumú fájl, amit futás közben tudunk a – már a memóriában lévő – kernelhez hozzácsatolni. A kernel modulok szépen kiegészítik a Linux rendszereket azzal, hogy csökkentik a kernel méretét és a rendszer betöltésének idejét úgy, hogy bizonyos ritkábban használt szolgáltatásoknak csak akkor kell "betöltődniük", amikor valóban szükség van rájuk. Ezek a ritkán használt szolgáltatások általában a hardvereszközök vagy fájlrendszerek meghajtóihoz tartoznak. Egy másik előnye a moduloknak, hogy a kernel bővítését sokkal rugalmasabbá teszik, ugyanis a kerneltől jól szeparáltan használhatók és fejleszthetők. Egy új modul teszteléséhez még a rendszer újraindítása sem szükséges.

Általában a kernel fordításakor bizonyos szolgáltatások kapcsán ki lehet választani, hogy a szolgáltatást bele kívánjuk-e linkelni a kernelbe, vagy modulként szeretnénk-e telepíteni. Érdekes lenne az új szolgáltatást is modulként megvalósítani, viszont nagy akadály, hogy egy kernel modul nem férhet hozzá a kernel összes változójához, pontosabban a kernel ELF² szimbólumaihoz.[4] Csak azokhoz férhet hozzá, amelyek a kernel forráskódjában határozot-

¹Léteznek kernel szintű processzek – kernel thread, kernel szál – is, amik viszont közvetlenül elérik a kernel által használt változókat és eljárásokat.

²Executable and Linkable Format

tan exportálva lettek. A rendszer számos fontos változója nincs exportálva, így modulból ezek exportálása nélkül nem tudnánk a lekérdezéseket elvégezni. Lehetséges a változók exportálása, ám ez a kernel több forrásfájljának módosításával járna, ami kisebb változtatások, új funkciók hozzáfűzése esetén elkerülendő.

3.2. A szolgáltatás típusa

Összefoglalva : adatokat szeretnénk a kernelből egy felhasználói processznek átadni. Több processz is használhatja egyszerre a szolgáltatást, és valószínűleg nagyobb adatmennyiség mozgatására lesz szükség. Ennek megvalósítására számos lehetőség jöhet szóba.

rendszerhívás : a kernellel való kapcsolattartás legalapvetőbb és legegyszerűbb formája.

bejegyzés a proc fájlrendszeren : akár egy új bejegyzést is készíthetünk a proc fájlrendszeren, ami abban különbözik a többitől, hogy azt olvasva az összes általunk kívánt adatot megkapjuk.

kernel szál : a kernel több külön folyamatból is állhat. Szép megoldás lenne a feladatot egy kernel modul létrehozásával megoldani, ami egy – vagy akár több – kernel szálon keresztül hajtja végre a szolgáltatást, mivel így nem lenne szükség a kernel komolyabb módosítására. Ebben az esetben egy szolgáltató folyamat létezne, amely kérésekre átadja a kért adatokat. Külön téma lenne, hogy milyen eszközökkel kommunikáljon egymással a kernel szál és a szolgáltatást igénylő folyamat. Ez a módszer a nem exportált változók miatt további változtatásokat igényelne.

3.3. Az adatátvitel típusa

Ha nem a hagyományos proc megoldást választjuk, külön kell megválasztanunk az adatátvitel módját is. Egyszerű másolásról a memóriában ugyanis nem lehet szó, ám ennek megértéséhez egy másik ismeret szükséges : egy processz két féle kontextusban, felhasználói illetve kernel kontextusban futhat. Általában, ha egy processz valamilyen szolgáltatást kér a kerneltől, a processz átlép kernel kontextusba. Kernel kontextusból viszont nem lehet egyszerűen elérni a felhasználói kontextushoz tartozó címetet.³

Ez a probléma könnyen megoldható a `copy_to_user()` és `copy_from_user()` eljárásokkal, bár ezen kívül több említésre méltó módja létezik a kernel és a felhasználói folyamatok közti adatátvitelnek. Ezek közé tartoznak a netlink socketek, speciális eszközfájlok, és üzenetsorok, melyek egy kényelmes interfészt nyújtanak a folyamatok közti adatátvitelre.⁴ Vizsgálatok szerint az osztott memória és az üzenetsorok alkalmasak a kernel és felhasználói kontextus közötti leggyorsabb adatátvitelre, viszont további módosításokat igényelnek a kernelben. [5] Bár e módszerek között nem elhanyagolható sebességkülönbség van, a hagyományos proc féle módszernél mind nagyságrendekkel gyorsabbak, tehát ezeket alkalmazva mindenképpen jobb teljesítményre számíthatunk.

³Címtéren egy adott instrukciófolyam által használható virtuális memóriarekeszek halmazát értjük.

⁴Ebből az is következik, hogy ezek egy szolgáltató folyamat létezését követelik meg.

Az adatátvitel nehezebben megoldható problémája abban rejlik, hogy a rendszer állapotát jellemző adatok mennyisége – a processzek számának dinamikus voltából fakadva – változó, azaz előre nem lehet eldönteni, hogy a memóriában mennyi helyet fognak foglalni. A kérdés tehát az, hogy hogyan készítsünk egy szolgáltatást és hozzá kapcsolódó "kliens" programokat úgy, hogy előre nem ismert méretű adattömeget is gond nélkül tudjanak cserélni egymás között. Az üzenetsorok, netlink socketek, és speciális fájlok erre a problémára működésüknél fogva egy kényelmes megoldást nyújtanak, az imént felsorolt többi adatátviteli mód viszont önmagában alapvetően nem alkalmas ennek kezelésére, ezért ezekre külön megoldást kell találni.

Egy lehetséges megoldás, hogy megtelt puffer esetén a kernel átteszi azt a felhasználói processz memóriájába, és a következő hívás esetén pedig majd az előzőleg megmaradt adatokat teszi át, viszont ehhez bonyolult nyilvántartási módszerre lenne szükség, főleg, ha egy időben több folyamat is használná a szolgáltatást.

Ha nem az előzőleg felsoroltakat használjuk, akkor dinamikus memória allokálásra lesz szükség. A kernelben a `kmalloc()` eljárás használható memória allokálására, a felhasználói folyamatokban pedig a `malloc()` és a `calloc()` eljárások. A `kmalloc()` használatával a szolgáltatás mindig tudna akkora tömbbel dolgozni, amelybe az összes információ belefér. A hívó folyamat viszont előre nem tudná, hogy mekkora méretű memóriára lenne szüksége, így lehet, hogy a rendszerhívás nem tudná az összes adatot átadni. Így a rendszerhívás és a folyamat között megint több híváson keresztül húzódó kommunikációra lenne szükség, ami túlságosan megbonyolítaná az implementációt.

Hogy a hívó folyamatnak egyáltalán ne kelljen memória allokálással foglalkoznia, a kernelben kell a felhasználói folyamat számára is memóriát allokálnunk. Ehhez a memória régiók ismerete szükséges. A kernel egy processz címtartományát régiókra osztja. Minden régió valamilyen szemponból összetartozó adatokat tartalmaz, tipikusak a `text`, `data`, és `bss` szegmenseknek megfelelő régiók, illetve a `heap`, melyből a processz a `malloc()` és `calloc()` eljárásokkal szabad memória területeket allokál magának. A memória régiók gyakran egy-egy fájl tartalmazznak magukban – azaz a fájl tartalma teljes egészében, vagy részben a memóriában helyezkedik el. A processzek saját maguk is létrehozhatnak új, tetszőleges méretű memória régiókat az `mmap()` eljárással. A kernelen belülről erre a feladatra pedig a `vm_mmap()` eljárás használható. Tehát a `vm_mmap()`⁵ segítségével lehet a kernelen belülről a felhasználói címtérben is memóriát allokálni. Általában egy processz a címtérnek⁶ csak a töredékét használja, tehát általában van hely nagy mennyiségű memória allokálására. Ha viszont nem csak egy lekérdezés erejéig fut, akkor – a helytakarékosság kedvéért – az allokált memóriaterületet a lekérdezés után érdemes felszabadítani. A memória régiókat a felhasználói kontextusból az `munmap()`, kernel kontextusból pedig a `vm_munmap()` eljárásokkal lehet felszabadítani.

⁵ A 3.4-nél korábbi verziójú kernelekben `do_mmap()` és `do_munmap()`.

⁶ Általában egy processz címtere 3 Gbyte.

3.1. táblázat. Példa egy processz – jelen esetben a `cat` – által használt memória régiókra. A `/proc/self/maps` lerövidítve.

08048000–08052000	r-xp	/bin/cat
08052000–08053000	rw-p	/bin/cat
09f86000–09fa7000	rw-p	[heap]
b73b6000–b75b6000	r--p	/usr/lib/locale/locale-archive
b75b6000–b75b7000	rw-p	
b75b7000–b76f7000	r-xp	/lib/i686/cmov/libc-2.11.3.so
b76f7000–b76f8000	---p	/lib/i686/cmov/libc-2.11.3.so
b76f8000–b76fa000	r--p	/lib/i686/cmov/libc-2.11.3.so
b76fa000–b76fb000	rw-p	/lib/i686/cmov/libc-2.11.3.so
b76fb000–b76fe000	rw-p	
b7711000–b7713000	rw-p	
b7713000–b7714000	r-xp	[vdso]
b7714000–b772f000	r-xp	/lib/ld-2.11.3.so
b772f000–b7730000	r--p	/lib/ld-2.11.3.so
b7730000–b7731000	rw-p	/lib/ld-2.11.3.so
bfd26000–bfd3b000	rw-p	[stack]

3.4. Szűrés

Az alternatív módszer kidolgozásánál érdemes ott gyorsítani a lekérdezést, ahol tudjuk, hogy egy jó ideig ne legyen a sebességgel probléma. Valamennyi sebességnövelést el lehet érni az adatok kernelen belüli szűrésével is. Így ugyanis a felhasználói processz címtartományába való másoláskor kevesebb adatra lesz szükség, bár, ahogy előzőleg említettem, a memóriaátvitel önmagában is elég gyors, tehát a jelentős sebességnövekedést nem ez fogja okozni. Mindenesetre érdemes lehet szűrési lehetőséget is belefoglalni az alternatív szolgáltatásba. Túl sok féle adatot nem érdemes szűrni, mert egy adott adatmennyiség és egy annál kicsit nagyobb adatmennyiség esetén a memóriaátviteli különbség gyakorlatilag jelenléktelen. Érdemesebb néhány, adatok egész nagy csoportját kiválasztó szűrési feltételt elhelyezni a szolgáltatásban.

Például

- csak valamilyen meghatározott erőforráskorlátot átlépő processzek listázása.
- választható, hogy a processzekhez a szolgáltatás gyűjtsön-e nyitott fájlokról adatokat – általában ezek összegyűjtése sok időt vesz igénybe.

3.5. Biztonság

Minden processzhez tartozhatnak olyan információk, melyeket csak illetékes processzek ismerhetnek. Amennyiben ezek az információk nem megfelelő kézbe kerülnek, nem csak a rendszeren belül keletkezhetnek gondok, de akár egyes személyeket is károk érhetnek. Ilyen szempontból fontos annak a biztosítása, hogy a szolgáltatás ellenőrzéseket végezzen a hívó

processz engedélyein, és annak megfelelően szolgáltatson adatokat a rendszerről. Például mindenképpen fontos, hogy egy processz nyitott fájljait ne láthassa mindenki – hiszen így mások tudomást szerezhetnek olyan fájlokról, aminek a jegyzékére nincsen olvasási jogosultságuk. De célszerűbb arról gondoskodni, hogy egy átlagos felhasználó más felhasználók által futtatott processzekről csak nagyon korlátozott információkat, vagy semmit se tudhasson meg. Ezek biztosítására is külön gondot kell fordítani egy új szolgáltatás megalkotásakor.

3.6. Az adatok összegyűjtése

Az utolsó lényeges szempont az, hogy honnan és hogyan gyűjti össze a szolgáltatás az adatokat. Általában minden adat csak egy helyen található meg a kernelen belül, így ennek megválasztásában nincs sok lehetőség.

A processzenkénti adatok mind egy helyen foglalnak helyet : processz leírókban.⁷ Ez egy olyan struktúra, ami egy processzre vonatkozó összes információt, vagy információt tartalmazó más struktúrákra való mutatót tartalmaz.

A processz leírók végiglátogatására a Linux kernelben egy kényelmes eszköz – tulajdonképpen egy makró – létezik, a `for_each_process()`, ami azt használja ki, hogy mindegyik processzhez tartozó processz leíró egy láncolt listára van felfűzve, melynek kiindulópontja a rendszer által indított első folyamat, az `init` leírója.

```
#define for_each_process(p) \
for (p = &init_task ; (p = next_task(p)) != &init_task ; )
```

A rendszer egészére vonatkozó adatok elszórtan helyezkednek el, általában mindegyik egy globális változó formájában, vagy az adott alrendszerhez tartozó struktúra egy elemében. Általában mindegyikhez külön eljárás létezik, mely az adott adatok előszedése mellett a konkurens hozzáféréshez szükséges műveleteket is elvégzi.

Külön említést érdemelnek a konkurens hozzáférés által megkövetelt intézkedések (például lock-ok kérése, counterek növelése), bár ezek nem opcionális dolgok, hanem feltétlenül meg kell őket megvalósítani.

Persze a kernel is csak azokat az információkat tudja összegyűjteni, amik léteznek. Ez alatt azt értem, hogy léteznek olyan adatok, melyeket olyan komponensek kezelnek, amik nem feltétlenül lettek belefördítva a kernelbe. Ha ezek nem részei az aktuális kernelnek, azaz fordításkor nem lettek kiválasztva, akkor az általuk kezelt adatok értéke valamilyen semleges érték lesz, általában 0 vagy NULL, de valószínűbb, hogy az eljárásokkal együtt az adott változók és mezők sem lettek részei a kernelnek. A kernel forrásában minden konfigurálható lehetőséghez egy makró tartozik. Például `CONFIG_TASK_DELAY_ACCT` jelzi, hogy a kernelbe bele lett-e fordítva az a komponens, amely a folyamatok által a rendszer egyes erőforrásaira várakozással eltelt időt jegyzi fel. Ennek függvényében a forráskódot, ami ezt használja, két részre kell osztanunk.

```
#ifdef CONFIG_TASK_DELAY_ACCT
//az összegyűjtött adatokkal való műveletek végzése
```

⁷A Linux kernelben process descriptor-nak nevezik és `struct task_struct`-ként van definiálva.

```
#else
//nincsenek összegyűjtött adatok, ennek megfelelő műveletek végzése
#endif
```

3.7. Szükséges adatok

Eddig volt szó a rendszerállapot lekérdezés hagyományos módjáról, a benne rejlő előnyökről és hátrányokról, és röviden rendszereztem, hogy milyen lehetőségek vannak egy rendszerállapot lekérdezési szolgáltatás elkészítésekor. Egyedül arról nem írtam még részletesebben, hogy milyen adatok azok, amikre általában monitorozáskor szükség van. Most ezek közül sorolom fel a legfontosabbakat.

processzor terheltség : azt jellemzi, hogy egy adott időintervallumban a folyamatok összesen mennyi ideig használták a processzort, vagy processzorokat. Másképpen megfogalmazva a processzor kihasználtságát jellemzi. Az összes processz futási idejének összegzésével lehet kiszámolni.

szabad memória : a rendszer szabad memóriája, más néven RAM, bájtokban, vagy lapokban⁸ mérve.

swap adatok : ha a RAM megtelt, akkor a rendszer egy tároló eszközön kijelölt fájlba, vagy partícióba helyezheti el az éppen nem használt lapjait. Általában, amikor ennek használatára szükség van, erős teljesítménycsökkenésre számíthatunk.

hálózati eszközök és számlálók : milyen hálózati eszközök üzemelnek a rendszerben, és milyen forgalommal, azaz eddigi küldött bájtok, fogadott bájtok, sikertelenül küldött bájtok, stb.

perifériás eszközök és számlálók : a hálózati eszközökhöz hasonló jellegű adatok.

Processzenkénti adatok :

állapot : egy processz több fajta állapotban lehet : futásra kész, megszakítható, nem megszakítható, megállított, zombi, stb.

pid, ppid : a processz azonosítója és a processz szülőjének – az őt létrehozó processznek – az azonosítója.

tgid : léteznek olyan programok, melyek futás közben több szálból állnak, azaz több folyamat tartozik hozzájuk egyszerre. Ezek együtt egy csoportot alkotnak, melynek azonosítója is van, a thread group id.

uid : a folyamatot futtató felhasználó azonosítója.

programnév : a programnak a neve, ami a folyamat instrukciófolyamába lett betöltve.

indítás ideje

⁸egy lap általában 4 Kbyte.

stime és utime : kernel és felhasználói kontextusban eltöltött – azaz működéssel eltöltött – idők.

prioritás : a kernel a processzek között a folyamatosság látszatának fenntartása céljából egy adott algoritmus szerint váltogat. Egy magasabb prioritású folyamat a többihez képest összességében nagyobb eséllyel fog a váltogatások alkalmával futni.

perifériahasználat : Mennyi a processz eddigi összes I/O forgalma

memóriahasználat : Minden processznek egy saját virtuális címtere van. Az azonos `tgid` értékű processzeknek viszont nem. A legfontosabb ennek az esetében is a használt, illetve szabad bájtok száma. Fontosak lehetnek a memória régiók adatai is.

nyitott adatfolyamok : a processzhez tartozó nyitott adatfolyamok.⁹ Külön gondot okoz, hogy előre nem lehet tudni, hogy egy processznek hány nyitott adatfolyama van, illetve, mivel ezeknek nevük is van, ismét a változó adatmennyiség átvitelének problémájával kell számolnunk.

⁹Egy nyitott fájlhoz egy nyitott adatfolyam tartozik.

4. fejezet

Az új szolgáltatás

A dolgozat ettől a ponttól fogva konkrétabbá válik, és az általam készített új szolgáltatásról fog szólni. Munkám során a Linux kernel 3.6.2.-es verziójával dolgoztam. A működés más verziójú kernelek esetén nem garantált, de minimális mennyiségű változtatással a szolgáltatás azokba is átültethető.

4.1. Alapvető tulajdonságok

A szolgáltatást rendszerhívásként valósítottam meg, a quickstats nevet adtam neki, és a 222-es számot kapta meg.¹ Azért rendszerhívásként valósítottam meg, mert a kapcsolat megteremtése egy processz és a kernel között így a legegyszerűbb, és az általam választott megoldás a memória kezelésére – `vm_mmap()` használata – így volt könnyen megvalósítható.

```
/*  
 * dd a hívó processz címtérében lévő struktúrára mutat.  
 * A kernel ezen keresztül fogja az általános információkat  
 * és a többi struktúrához tartozó mutatókat átadni.  
 */  
asmlinkage long sys_quickstats(struct qs_data_desc *dd, long flags);
```

A rendszerhívás teljes mértékben megoldja a változó adatmennyiséget azzal a megoldással, hogy a kernelen belül allokal memóriát a hívó folyamat címtérében is. Így a hívó folyamatnak nem kell az adatok mennyiségéről becsléseket tennie, nem lehetséges az az eset, hogy túl kevés memóriát foglal, és nem kell a rendszerhívást többször meghívnia a összes adat átviteléhez. A monitorozó programok általában rövid élettartamúak, így akármekkora memória régiót is allokal a kernel a folyamat számára, az nem lesz különösebb hatással a folyamat életére, hacsak nem hosszabb távon kíván működni a folyamat. Ebben az esetben érdemes a lefoglalt területet felszabadítani, nehogy később túl kevés szabad memóriája legyen a processznek, amikor többre lenne szüksége. Erre a feladatra alkalmas az `munmap()` eljárás.

¹Minden rendszerhíváshoz tartozik egy szám, amivel a hívó folyamatok azonosítják.

```

userptr = (struct qs_short_p_desc*)vm_mmap(NULL, 0,
                                           alloc_size + str_size,
                                           PROT_READ | PROT_WRITE,
                                           MAP_PRIVATE | MAP_ANONYMOUS, 0);

```

A `sys_quickstats` csak akkor nem sikeres, ha a két kontextus közül valamelyik számára nem sikerült megfelelő mennyiségű folytonos címtartományt lefoglalni – azaz egyébként is nagy a baj, mert túlságosan sok processz van, a processzek túlságosan sok fájlt használnak, vagy túl kevés szabad memóriája van a rendszernek. Ebben az esetben a visszatérési értéke `ENOMEM`.

A rendszerhívás képes az egyes folyamatokhoz tartozó nyitott adatfolyamok adatainak összegyűjtésére is. Mivel a string-ek helyigénye előre meghatározhatatlan, de egyébként is biztosan több néhány bájt nál, ezeket egy külön pufferben gyűjti össze. Ez a folyamat egy jelentős részét teszi ki a rendszerhívásnak, így a `flags` argumentumban jelezhető, hogy a hívó processz kíváncsi-e ezekre. Ha nem, akkor a rendszerhívás nem gyűjti össze a neveket, így időt spórolva meg.

```
#define QS_NOFILES 1
```

A lekérdezés során az eljárás két tömböt allokal. Egyet a processzek rövidebb jellegű – azaz néhány bájtnyi terjedelmű – adatainak, egy másikat pedig a processzhez tartozó nyitott fájlok és a nevük számára. A rendszerszintű adatok egy statikus struktúrába kerülnek.

```

/* ebbe kerülnek a rendszerszintű adatok és a két tömb kezdőcíme */
struct qs_data_desc kernel_dd;
...
/* puffer a rövid processz információknak */
kernelptr = (struct qs_short_p_desc*)kzalloc(alloc_size, GFP_KERNEL);
...
/* puffer a fájlneveknek */
str_size = alloc_size;
strptr = (struct qs_filename*)kzalloc(str_size, GFP_KERNEL);

```

A feltöltött puffereket `copy_to_user()` -el másolja át a felhasználói pufferbe.

```

/*
 * A kernelptr es strptr egymás után lesznek elhelyezve a
 * hívó processz memóriájában
 */
retval = copy_to_user((void*)userptr, (void*)kernelptr, alloc_size);
...
retval = copy_to_user((void*)userptr+alloc_size,
                     (void*)strptr, str_size);

```

A hívó processz csak akkor kap meg minden processzre vonatkozó információt a szolgáltatástól, ha rendelkezik a `CAP_SYS_ADMIN` jogosultsággal, egyébként csak a saját processzeit vizsgálhatja.

```

for_each_process(t) {
if(!has_capability(current, CAP_SYS_ADMIN) &&
    (__kuid_val(t->cred->uid) != __kuid_val(current->cred->uid)))
    continue;
    ....
}

```

4.2. Adatstruktúrák

A rendszerhívás működése alapvetően három struktúrán nyugszik.

A `struct qs_data_desc` egyben a rendszerre vonatkozó átfogó információkat tartalmazza, másrészt pedig a processzenkénti információkat tartalmazó tömb és a fájlneveket tartalmazó tömb kezdőcímeit.

```

#define NETDEV_ARRAY_LENGTH 32

struct qs_data_desc{
    struct qs_short_p_desc *ptr; /* a processzenkénti adatok */
    struct qs_filename *str_ptr; /* fájlneveket tartalmazó tömb eleje */
    int nr_processes;           /* processzek száma */
    long data_size;             /* a létrehozott memória régió mérete*/
    u64 io_read, io_write;      /* összes I/O forgalom bájtokban */

    /* Halozati eszközök információi */
    struct qs_netdev_info netdev_array[NETDEV_ARRAY_LENGTH];
};

struct qs_netdev_info{
    __u64 rx_packets;           /* összes fogadott csomag */
    __u64 tx_packets;           /* összes küldött csomag */
    __u64 rx_bytes;             /* összes fogadott bájt */
    __u64 tx_bytes;             /* összes elküldött bájt */
    __u64 rx_errors;            /* fogadási hibák */
    __u64 tx_errors;            /* küldési hibák */
    __u64 rx_dropped;           /* eldobott bejövő csomagok */
    __u64 tx_dropped;           /* eldobott kimenő csomagok */
    __u64 multicast;            /* multicast csomagok */
    __u64 collisions;           /* ütközések */
};

```

A `struct qs_short_p_desc` egy "rövid" processz leíró, mivel az eredeti processz leírónak csupán a leglényegesebb elemeit tartalmazza.

```

struct qs_short_p_desc{
    volatile long state;          /* a processz állapota */

    int prio;                     /* "dinamikus" prioritás */
    int static_prio;              /* nice érték */
    int normal_prio;              /* a várható prioritás */
    int rt_priority;              /* real-time prioritás */
    unsigned int policy;          /* a folyamat ütemezési stratégiája */
    long pid;
    long tgid;
    long ppid;
    cputime_t utime, stime;
    long start_time;
    long uid;
    char comm[TASK_COMM_LEN];     /* a folyamatot elindító parancs */
    long vm_size;                 /* virtuális memória mérete bájtokban */
    long open_fds;                /* nyitott fájlok száma */

    /*
     * a fájlneveket tartalmazó tömbben egy offset, ahol
     * a processzhez tartozó nyitott fájlok rekordjai kezdődnek
     */
    long f_names;

    /*
     * CONFIG_TASK_XACCT-al működik.
     * a processz összes io mennyiségét tartalmazza, byte-okban
     */
    u64 read_bytes, write_bytes;
};

```

Végül `struct qs_filename` a nyitott adatfolyamok neveit tartalmazza. A rendszerhívás ebben az esetben is egy dinamikusan allokált tömböt használ, de mivel a fájlnevek hosszúsága változó, a tömbben – ha ebben az esetben még nevezhetjük tömbnek – két ilyen struktúra között a hely változó hosszúságú lehet.

```

struct qs_filename{
    int desc;
    char name[1];
};

```

A hívó processz a tömb elejéből kiindulva úgy tudja megtalálni az aktuális struktúra után következőt, hogy `sizeof(int)` -et, a tömb aktuális elemében lévő sztring hosszát és még 1-et – a sztring végén a nulla karakter miatt – ad az aktuális struktúra címéhez.

```
struct qs_filename *ptr;
...
ptr = (struct qs_filename*) (ptr + sizeof(int) + strlen(ptr->name) + 1);
```

4.3. Az implementáció részletei

A quickstats rendszerhívás dinamikusan foglal helyet a `qs_short_p_desc` puffer számára, hogy bármennyi processz esetén elegendő hely legyen a feltöltésére. A helyfoglaláskor a rendszerben lévő processzek számát a `nr_processes()` eljárás segítségével tudja meg. Mivel az eljárás meghívása és a puffer feltöltése között még létrejöhetnek újabb processzek, a rendszerhívás a puffer allokálásakor a jelenleginél néhányval több processz számára foglal helyet.

```
#define EXTRA_SPACE 128
...
int pr_count = nr_processes;
...
alloc_size = pr_count + EXTRA_SPACE
```

Ha 128-nál is több új processz létezik, akkor azok már nem fognak beleférni a pufferbe, egyszerűen kihagyja azokat, és majd visszatéréskor jelzi, hogy megtelt a puffer. A processzek számára az ellenőrzés a `for_each_process` ciklusban történik.

A processzek száma változó, a hozzájuk tartozó nyitott adatfolyamok száma is változó, és az adatfolyamokhoz tartozó fájlok nevének hossza is változó, így a `qs_filenames` tömb méretének megbecsülése lehetetlen. A `sys_quickstats` eredetileg ugyanannyi memóriát allokál ennek a tömbnek, mint amennyit a `qs_short_p_desc` tömbnek rendelt. A tömböt addig tölti fel, amíg végig nem ment az összes processzen, vagy amíg az megtelt. Az utóbbi esetben egy kétszer akkora allokál, és az eddiginek a tartalmát annak az elejébe másolja, majd folytatja a munkáját.

A processzek által írt és olvasott bájtok számára vonatkozó adatok csak akkor érhetők el, ha a kernel fordításakor a `CONFIG_TASK_XACCT` opció ki lett választva. Ez az opció a processzek adatainak részletesebb rögzítését, többek között az írt és olvasott bájtok számának rögzítését tartalmazza. Ha az opció nem volt kiválasztva fordításkor, akkor a megfelelő mezőkbe 0 értékek kerülnek.

4.4. Példaprogram

A quickstats rendszerhívás használatára egy példaprogramot készítettem, amely hasonló jellegű információkat gyűjt össze és ír ki az általános kimenetére, mint a `ps` program, de még a `qs_data_desc`-ben lévő rendszerre vonatkozó információkat és a processzek nyitott fájljait is az általános kimenetére írja.

Alapvetően minden processz adatait egy külön sorba írja ki, és ugyanabba a sorba írja a processzek nyitott fájljait is. A program indításakor a `--sep_files` opció használata esetén a fájlneveket az őket használó processz sora után külön sorokban írja ki.

A program nem gyűjti össze a nyitott fájlokat a `--nofiles` opció használata esetén. Ekkor a rendszerhívásnak átadott flag argumentumban a `QS_NOFILES` bit be lesz állítva.

A programot `qstest`-nek neveztem el.

5. fejezet

Teszteredmények

Természetesen az új rendszerhívás csak akkor fogja megállni a helyét, ha a gyakorlatban is jól működik. Ebben a fejezetben a hagyományos programok és a quickstats rendszerhívást használó példaprogram vizsgálatának eredményeit fogom közölni.

A futtatási eredmények a várakozásaimnak megfelelően sokkal rövidebb futási időt mutattak a hagyományos módszerhez képest. A futtatásokat szándékosan többféle, szimulált környezetekben végeztem. Egy normális eseten kívül a többi mind valamilyen fokozott terhelést szimulált a rendszeren, melyben a két lekérdezési módszer közti különbség jobban meglátszott. Minden környezetben tízszer végeztem el a méréseket, a táblázatokban feltüntetett értékek a 10 vizsgálat eredményeinek átlagolt értékei.

A méréseket egy Intel Pentium 4 típusú 3.06 GHz-es processzoron futó rendszeren végeztem, melyen átlagosan 60 processz fut és a processzor átlagos kihasználtsága 1%.

5.1. Használt eszközök

Minden programnál két eszközt használtam a megfigyelésre. Egyrészt a futási időt szerettem volna megmérni, másrészt pedig a program által végzett rendszerhívások számát. Ugyanis a rendszerhívások túlságosan nagy száma az, ami miatt az alternatív módszer elkészítése szükséges volt. A használt eszközök a következők :

time : ez a bash parancsértelmező egy beépített eljárása. Az utána megadott parancs végrehajtását végigkíséri, és a befejeződésekor annak futási idejét közli – összes futási idő, felhasználói kontextusban eltöltött idő és kernel kontextusban eltöltött idő.

strace : ez a program az argumentumában megadott program végrehatása után annak a rendszerhívásairól statisztikákat ír ki. A tesztelés során a `-c` opciót használtam, ami minden rendszerhíváshoz kapcsolódóan megszámlolja, hogy hányszor használta az adott program.

Linuxon általában a programok a felhasználónak valamilyen információt adnak át, és ez mindenképpen igaz a monitorozó programokra is. A UNIX alapelveinek megfelelően minden ilyen program alapvetően az általános kimenetét használja. Ha Linuxon egy program kimenetét átirányítjuk a `/dev/null` speciális fájlba, akkor az sehol sem fog megjelenni, így csökkenthetjük a program végrehajtásához szükséges időt. Ezért a pontosság érdekében a

tesztelések során minden program kimenetét a `/dev/null` fájlba irányítottam át, azaz azt tudhatjuk meg, hogy az adatok előkerítése a kernelből mennyi időbe került, nagyjából függetlenül attól, hogy az adott program mit csinált velük.¹ Továbbá minden program esetében az összesített rendszerhívások számából kivontam a `write()` rendszerhívások számát, ugyanis ezek a programok csak az általános kimenetre írnak.

5.2. Vizsgált programok

Összesen 4 vizsgálatot végeztem minden szimulált környezetben.

- A `ps`, mint említettem, a monitorozási célra legáltalánosabban használt program, ezért a vizsgálatok többek között tartalmazzák a `ps -ef` futási adatait. Az `-ef` opció használatával a `ps` minden folyamatról részletes információkat gyűjt.
- A `ps-el` szemben vizsgálatokat végeztem a `qstest --nofiles` futtatásán is.
- Egyes esetekben hasznos lehet tudni, hogy bizonyos fájlok mely folyamatok által vannak használva. Erre az általánosan használatos program az `fuser`, melynek argumentumában fájlneveket megadva azokról a folyamatokról ír ki információt, amelyek a megadott fájlok valamelyikét használják. Ennek a programnak a működésére is végeztem méréseket, melyek során az `fuser TDK.tex` futását vizsgáltam.
- Bár a `quickstats` rendszerhívás egyelőre nem gyűjt olyan részletes információkat a használt fájlokról, mint az `fuser` program, bizonyos esetekben mégis használható lehet, ezért ezt is használtam arra a feladatra, amelyre az `fuser` használatos, a következő képpen : `qstest | grep TDK.tex`. Ebben az esetben a rendszerhívások száma egyenlő a `qstest` által végzett rendszerhívások számával. Az összes rendszerhívásba nem számoltam bele a `grep` által végzett rendszerhívások számát.

5.3. Eredmények

5.3.1. Processzek számának növelése

Mint említettem, a legnagyobb gond a `proc` fájlrendszerrel az, hogy sok processz esetén sok rendszerhívásba kerül végighaladni a fájljain. Ezzel kapcsolatban végeztem méréseket.

¹Ugyanis nem feltétlenül az a cél, hogy minden információt a kimenetére írjon egy monitorozó program.

5.1. táblázat. Végrehajtás átlagos környezetben

program	végrehajtás ideje	rendszerhívások száma
ps -ef	0.008s	910
qstest --nofiles	0.001s	25
fuser TDK.tex	0.009s	1076
qstest grep TDK.tex	0.002s	25

5.2. táblázat. Végrehajtás 4000 processz esetén

program	végrehajtás ideje	rendszerhívások száma
ps -ef	0.901s	56884
qstest --nofiles	0.019s	25
fuser TDK.tex	2.006s	81015
qstest grep TDK.tex	0.080s	25

5.3. táblázat. Végrehajtás 8000 processz esetén

program	végrehajtás ideje	rendszerhívások száma
ps -ef	2.496s	112887
qstest --nofiles	0.077s	25
fuser TDK.tex	5.200s	161019
qstest grep TDK.tex	0.299s	25

5.3.2. Fokozott processzor terhelés esete

A programok futási idejét befolyásolja a fokozott processzor terhelés. Ugyanis ekkor a monitorozó programoknak osztozniuk kell más processzekkel a processzoron. A példaprogram ebben az esetben is jobban teljesít.

5.4. táblázat. Végrehajtás az átlagos környezet és 16 processzort igénybevevő processz esetén

program	végrehajtás ideje	rendszerhívások száma
ps -ef	0.148s	1120
qstest --nofiles	0.030s	25
fuser TDK.tex	0.191s	1334
qstest grep TDK.tex	0.052s	25

5.5. táblázat. Végrehajtás az átlagos környezet és 32 processzort igénybevevő processz esetén

program	végrehajtás ideje	rendszerhívások száma
ps -ef	0.359s	1344
qstest --nofiles	0.048s	25
fuser TDK.tex	0.550s	1638
qstest grep TDK.tex	0.070s	25

5.3.3. Kevert környezet

Olyan vizsgálatokat is végeztem, melyek az előző terhelések egyszerre való teljesülése esetén mentek végbe.

5.6. táblázat. Végrehajtás 4000 processz és 16 processzort igénybevevő processz esetén

program	végrehajtás ideje	rendszerhívások száma
ps -ef	8.804s	57204
qstest --nofiles	0.212s	25
fuser TDK.tex	19.563s	81464
qstest grep TDK.tex	0.890s	25

6. fejezet

Összefoglalás

A dolgozat során röviden bemutattam a Linux operációs rendszer egyedülálló megoldását a rendszerinformációk kezelésére, és azt, hogy miért van szükség egy ezt kiegészítő módszerre. A 3. fejezetben felvázolt lehetőségek alapján létrehoztam egy új rendszerhívást a Linux kernelben, amely az említett problémát hívatott megoldani. A mérések eredményei alapján megállapíthatjuk, hogy a bemutatott módszer sebessége mindenképpen többszöröse a hagyományosnak, sőt esetenként nagyságrendekkel gyorsabb, így a megoldás hasznossága magáért beszél.

Az általam kínált implementáció példaként készült, hogy a proc fájlrendszerrel szemben egy bizonyos szempontból hatékonyabb módszert bemutasson. Egy olyan implementációhoz, amely hivatalosan is része lehet a kernelnek, hosszú út vezet. A módszer sikeréért még intenzív tesztelésre és optimalizálásra van szükség, bár az is lehetséges, hogy később valamilyen más formában – nem rendszerhívásként – fog tovább létezni.

Egy ilyen szolgáltatás elsősorban a jelenleg is elterjedt rendszerfigyelő eszközök számára lenne igazán kedvező, így akkor nevezhetnénk igazán sikeresnek, ha ezek az eszközök, többek között a ps, top, fuser vagy lsof ezt a szolgáltatást használnák, így növelve a Linux rendszerek teljesítményét.

Az elkészített rendszerhívás és a rendszerhívást használó program forráskódjai elérhetők a <http://users.iit.uni-miskolc.hu/~peto5/munkak/> URL alatt.

Irodalomjegyzék

- [1] Dr.Vadász Dénes : *Operációs rendszerek* , Miskolci Egyetem, Általános Informatikai Tanszék, 2002
- [2] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2012
- [3] Daniel P. Bovet, Marco Cesati : *Understanding the Linux Kernel*, O'Reilly, 2005
- [4] "Chapter 4 : Object files, Chapter 5 : Program loading and dynamic linking" , *System V Application Binary Interface*, 1997
- [5] Ryan Slominski : *Fast Kernel/User Data Transfer*, College of William & Mary, 2007
- [6] The Linux Documentation Project, <http://tldp.org>
- [7] Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini : *Linux Device Drivers, 3rd Edition*, O'Reilly, 2005
- [8] Linux Cross Reference, <http://lxr.free-electrons.com>