MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR

TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Alternatív processz állapot és statisztika lekérdezési módszer a Linux kernelben

Piller Imre
Mérnök informatikus MSc, I. évfolyam

Konzulens:
Vincze Dávid
egyetemi tanársegéd
Általános Informatikai Tanszék

MISKOLC, 2011

Tartalomjegyzék

1. fejezet

BEVEZETÉS

A GNU Linux egy igen elterjedt operációs rendszer. Számos architektúrát támogat, és a kernel forráskódja is szabadon hozzáférhető. Manapság már asztali gépeken is használják, de még mindig túlnyomórészt a kiszolgálókon, kiszolgálórendszereken van jelen. Különféle alkalmazások futtathatók rajta, amelyek többnyire több processzt is használnak működésük során. Webszerverek, adatbázisok esetében például a több ezer párhuzamosan futó processz is gyakori jelenség.

A gyakorlati tapasztalatok azt mutatják, hogy leterhelt, időszakosan túlterhelt kiszolgálók esetén ezeknek a figyelése, monitorozása nehézkes lehet. Alkalmanként maguknak a processzeknek a monitorozása is figyelemreméltó erőforrásigényű, így elfogadhatatlan válaszidőket eredményez. Ennek a problémának az enyhítésére mutat be a dolgozat egy lehetőséget.

Érdemes tüzetesebben szemügyre venni a kernel felépítését és működését, még a változtatások, bővítések részletezése előtt. A kernel fejlesztése során mindig több kívánalmat is egyidejűleg szem előtt kell tartaniuk a fejlesztőknek, amik időnként ellentmondásosak, így fényt kell deríteni azokra az ok-okozati összefüggésekre, amik a konkrét implementációig elvezettek. Ez a dolgozat szempontjából különösen a processzek, illetve a modulok tekintetében érdekes.

A dolgozat első része a folyamatokat, illetve az azokhoz tartozó adatok tárolási módját tárgyalja. Ezek azok az adatok, amiket majd össze kell gyűjteni, struktúrába rendezni, és megjeleníteni, ezért fontos sorra venni, hogy mire szolgálnak, és hogyan lehet majd a későbbiekben ezeket lekérdezhetővé tenni.

A központi témát a proc fájlrendszer adja. Ez biztosítja a hátteret a rendszer monitorozására szolgáló jelenleg használatban lévő felhasználói programoknak. Ez egy interfész a rendszermag és a felhasználói tér között, ezért érdemes megnézni, hogy a kernelben hogy van megvalósítva, illetve hogy az adatok átvitele hogyan oldható meg ezen keresztül.

Jelenleg a processzek megfigyelése azt az elvet követi, hogy az adataik ebben a virtuális fájlrendszerben kapnak helyet, majd onnan a felhasználói programok azokat szöveges fájlok formájában dolgozzák fel. Ennek a kiváltására egy új, egyszerű interfész került definiálásra a kernel és a felhasználói tér között, amelynek segítségével minimális rendszerhívás használatával elérhetővé válik a szükséges információ. Szükség van még továbbá az interfészt használó felhasználói programokra is. A dolgozat bemutat egy elkészült alkalmazást is, amely segítségével az új interfészt használva a rendszer processzeiről információkat lehet kinyerni.

A dolgozat utolsó fejezetében néhány teszteset is szerepel, amely egy gyakran használt lekérdezés segítségével hasonlítja össze a jelenleg elterjedt lekérdezési mód hatékonyságát, a dolgozatban bemutatott új megoldással a rendszer különböző szintű terheltsége mellett.

2. fejezet

A PROCESSZEK NYILVÁNTARTÁSA

A programok a rendszerben folyamatokként hajtódnak végre. Ezekhez a futtatható kódon kívül további lényeges információk is hozzátartoznak, amelyek majd a folyamatok kontextusát adják, úgy mint például a regiszterek állapota, illetve a felhasznált memóriaterületek.

2.1. *A processz tábla*

Az operációs rendszer bizonyos megközelítésben arra szolgál, hogy a benne futtatott processzek számára környezetet biztosítson. Ehhez a processzekhez tartozó minden lényeges információt nyilván kell tartania. A Linux kernelben erre a `linux/sched.h` fejállományban található `task_struct` struktúradefiníció szolgál. Ezek láncolt listája adja majd a processz táblát, amiben a rendszerben aktuálisan jelen lévő összes processz megtalálható. A kernel forráskódja a *task* és a *process* kifejezéseket általában szinonímákként használja. Régebben a *task* elnevezés volt a gyakoribb, később a *process*, viszont a rendszer alapjául szolgáló részeknél, amelyek már a kezdeti verzióktól szerepeltek a kódban (mint a `task_struct` is), azoknál ez nem minden esetben került átírásra.

A `task_struct` struktúra elég terjedelmes, mivel minden információnak szerepelnie kell benne, ami az azonosításukhoz, memóriakezeléshez, szálkezeléshez illetve más feladatok elvégzéséhez szükséges. Jelenleg, a 3.0.4 verziójú Linux kernelben a struktúra mérete 1128 bájt. A későbbi feladatokra való tekintettel sorra kell venni, hogy milyen részeket is tartalmaz a struktúra, melyeket lehet majd célszerű felhasználni az állapot lekérdezése során, illetve a statisztikák összeállításához.

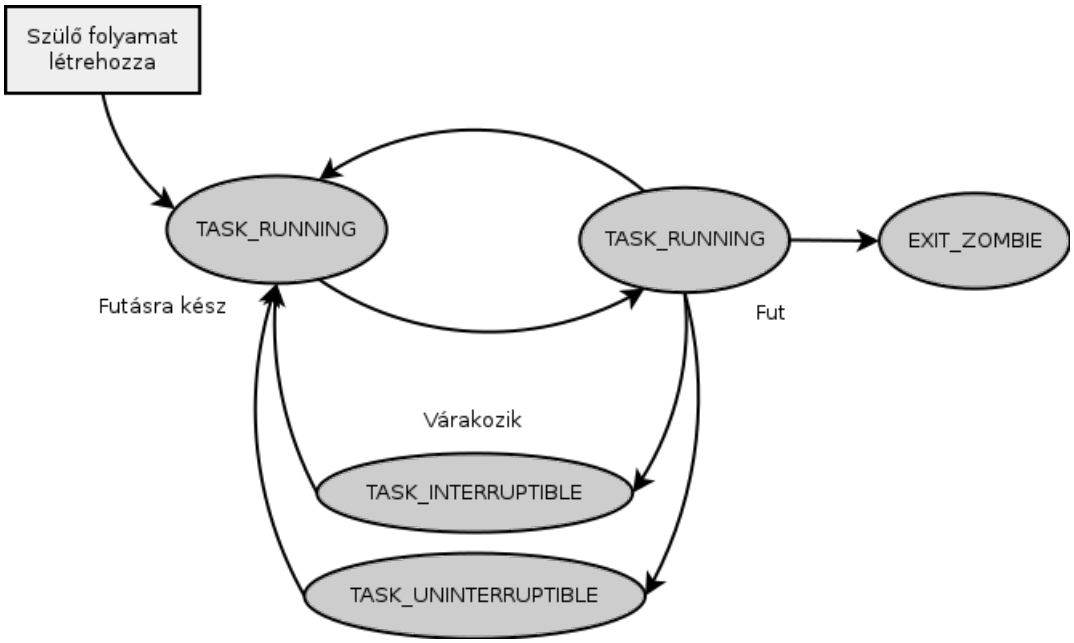
Minden processz számára szükséges egy egyedi azonosító (`PID` - process identifier). Ez a POSIX szabványnak megfelelően `int`-ként definiált. (A típus megadása két lépcsőben történik; a struktúrában `pid_t` típus szerepel, ami a `types.h` -ban `__kernel_pid_t`-ként, ami pedig a `posix_types.h`-ban, mint `int` típus van megadva. Ezt a többi mező esetében is hasonlóképpen oldották meg.)

A `state` mező a folyamat állapotát jelzi. A felvehető értékeinek definíciói szintén a `sched.h` állományban szerepelnek, így a folyamat mindig az alábbi állapotok valamelyikében kell, hogy legyen:

- `TASK_RUNNING`: A folyamat éppen fut, vagy futásra kész állapotban van.
- `TASK_INTERRUPTIBLE`: A folyamat egy esemény bekövetkeztére vár. Képes fogadni a jelzéseket, és ismét `TASK_RUNNING` állapotba kerülni.
- `TASK_UNINTERRUPTIBLE`: Hasonló az előző állapothoz, azzal a különbséggel, hogy nem kezeli a jelzéseket, mivel vélhetően egy kiemelt fontosságú feladatot hajt éppen végre a folyamat.

- TASK_STOPPED: A processz futását a kernel felfüggesztette egy SIGSTOP vagy SIGTSTP jelzés hatására. Ismét RUNNING állapotba kerülhet egy SIGCONT jelzés hatására.
- TASK_TRACED: A folyamat ebben az állapotban van, amíg más folyamatok figyelik a viselkedését (például hibakeresés céljából).
- TASK_KILLABLE: Egy viszonylag új állapot (a 2.6.25-ös verzióban jelent meg), amelyben a folyamat várakozhat. Leginkább a TASK_UNINTERRUPTIBLE állapotra hasonlít, azzal a különbséggel, hogy ezt speciális jelzések megszakíthatják. [5]
- EXIT_ZOMBIE: A folyamat már leállt, viszont a szülő processznek még bizonyos statisztikai információkra szüksége lehet (például a visszatérési értékre).
- EXIT_DEAD: A processzre már nincs szükség, csak az eltávolítására vár.

A fő állapotátmenetek a 2.1 ábrán láthatók.



2.1. ábra. A folyamatok fő állapotátmenetei.

A flags mező egyes bitjeivel az állapotjellemzők állíthatók be. A bitek jelentésére a PF_ kezdetű definíciók utalnak, például néhány a gyakrabban használtak közül:

- PF_STARTING: A folyamat létrejön.
- PF_MEMALLOC: Memóriát foglal le.
- PF_SIGNALED: Egy jelzést kapott.
- PF_EXITING: A folyamat leáll.

A folyamat visszatérési értékét az exit_code mező tárolja. Ezt az értéket fogja majd megkapni a szülőprocessz a folyamat befejeztével.

A folyamat elindításához kiadott parancsot a comm mező tárolja. Ez egy egyszerű karaktertömb maximálisan TASK_COMM_LEN hosszú parancs tárolására. (Ennek értéke a jelen implementációban 16.)

A folyamatok számára azt a látszatot kell kelteni, hogy nekik egy saját processzoruk van. Ennek a biztosítását az ütemező végzi, mégpedig úgy, hogy a processzeket adott időközönként váltogatja. Azért, hogy a rendszer működése hatékony legyen, az egyes feladatok különböző súllyal kapnak időszeleteket, méghozzá a prioritásuknak megfelelően. A processz jellemzői között természetesen ezek a prioritások is megjelennek, méghozzá a `prio` és `static_prio` mezők formájában. (A helyes ütemezés megválasztása egy igen összetett feladat. A jellemzők között további erre vonatkozó adatok is szerepelnek még. [9])

A folyamatok egymással hierarchikus kapcsolatban vannak, mivel minden folyamatot (az *init* folyamat kivételével, amely a rendszer indításakor jön létre) csak egy másik folyamat hozhat létre. Az így kialakult viszonyokat is tárolni kell. A `real_parent` mező a szülő processzre mutat. Ha a tényleges szülő processz időközben leállt volna, akkor az init processz veszi át a helyét. A `parent` szintén a szülőre mutat, viszont ez a `ptrace` rendszerhívás hatására megváltoztatható. A `children` és a `sibling` láncolt listák, amelyek a processz gyermekeire, illetve testvéreire mutatnak.

Többfelhasználós rendszereknél azt is fontos rögzíteni, hogy a folyamat melyik felhasználóhoz tartozik, illetve hogy ebből kifolyólag milyen jogosultságokkal rendelkezik. A két alapvető ezek közül a felhasználó, illetve a csoport azonosítója (`uid` és `gid`). Ezeknek van úgynevezett effektív változata is (`euid`, `egid`), amely általában megegyezik az előzőekkel, csak speciális esetekben vehet fel más értékeket.

A folyamatoknak memóriahasználatáról az `mm` és `active_mm` mezők adnak információkat [10].

A fájlrendszerhez a hozzáférést is processzek végzik, így nyilván kell tartani a fájlrendszerrel kapcsolatos információkat (`fs`), illetve azt, hogy mely processzhez mely megnyitott fájlok tartoznak (`files`) [11].

A folyamatok jelzéseket küldhetnek, vagy fogadhatnak. Ezek kezeléséhez további mezők tartoznak, mint például a `signal`, illetve a `sighand` [12].

2.2. *Processzek adatainak kezelése*

A processzek adatai az előbb bemutatott struktúrából közvetlenül is hozzáférhetőek, viszont célszerű az erre a célra szolgáló makrókat és függvényeket használni, mivel zárolásra illetve ellenőrzésekre lehet szükség. A `sched.h` rengeteg hasznos eszközt biztosít erre [3].

A processz azonosítója alapján gyorsan vissza kell tudni keresni az adatstruktúra mutatóját, az ütemező ezért azokat egy hash táblában is nyilvántartja (`pid_hash`). Ennek az implementációjához láncolt listát használ, amelyek kezelése mostmár egységes megoldást használ az egész kernelben.

A processzek listájának végigjárásához tartozik a `for_each_process` makró. Ezt használva elég a

```
struct task_struct* p;
for_each_process(p) {
        ...
}
```

formában megadni a ciklust, és ebben a `p` paraméter sorban be fogja járni a folyamatokat. A makró egy for ciklust takar, aminek a definíciója a következőképpen néz ki:

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

A `next_task` szintén egy makró:

```
#define next_task(p) \
    list_entry_rcu((p)->tasks.next, struct task_struct, tasks)
```

Hasonló makrómegadási módok több helyen is szerepelnek a kernelben. Ezekkel körültekintően kell bánni, mivel ha több egymásba ágyazott ciklust is tartalmazhatnak, így a `break` és `continue` utasítások nem feltétlenül azt a hatást fogják eredményezni, mint ami az egyszerűbb leírásmód miatt várható lenne. (Bizonyos helyeken ezért javasolják a `goto` használatát a ciklusból való kilépésre, mint például a `do_each_thread` esetében is.)

Az egyes állapotjellemzők maszkolásához is vannak definiált makrók, mint például a `task_is_stopped(task)` vagy `task_is_dead(task)`. Új állapotot is állíthatunk be a `set_task_state(tsk, state_value)` makróval, de ez inkább csak az ütemezőnek szükséges.

A makrók mellett rengeteg inline függvény is rendelkezésre áll, például:

- `rt_task(task)`: A `task` prioritását adja vissza.
- `task_pid(task)`: A folyamatazonosítóval tér vissza.
- `pid_alive(task)`: Ellenőrzi, hogy a struktúra érvényes-e még, mert ha nem, akkor a benne lévő mutatókat már nem szabad használni.
- `get_nr_threads(task)`: A `task` folyamathoz tartozó szálak számát adja meg.

Ezek a kód olvashatóságának javítása szempontjából lényegesek.

Az adatoknak van egy olyan része is, amelyek nem hozhatók kapcsolatba közvetlenül sem az ütemezéssel sem a memóriakezeléssel. Ezek tulajdonképpen csak információkat szolgáltatnak a működésről, de aktívan nem befolyásolják azt, vagy csak nagyon speciális helyzetekben van rájuk szükség, ezért nem érdemes azokat a processztáblában tárolni. Ezek a folyamatokhoz tartozó memóriaterületen kaptak helyet, esetenként szekvenciális fájlokon keresztül érhetők el, és jellemzően a proc fájlrendszer használja őket. Ilyen például a folyamat elindításánál kiadott parancs is az argumentumaival együtt.

3. fejezet

A PROC FÁJLRENDSZER

A Unix(-szerű) rendszerekben található egy pszeudó-fájlrendszer, aminek az a feladata, hogy a processzek adatait a felhasználói térben futó programok számára egyszerűen elérhetővé tegye. Ez a fájlrendszert a kernel már a bootolás során felcsatolja, általában közvetlenül a gyökérkönyvtárba (`/proc`).

A proc fájlrendszer a működés szempontjából nem létfontosságú, viszont ahhoz, hogy a rendszer aktuális működésével kapcsolatos információkhoz juthassunk, ahhoz elengedhetetlen. A kernel fordítása során a `CONFIG_PROC_FS` értékkel lehet engedélyezni a használatát (alapértelmezés szerint engedélyezve van). Linuxok esetében az ebben lévő fájlok néhány kivételtől eltekintve mind szövegesek, így az adatok kiolvasása, vagy beírása a hagyományos parancssori eszközökkel elvégezhető.

A fájlrendszer elsősorban a rendszer adatainak kiolvasásához volt használatos, viszont a szerkezete úgy van kialakítva, hogy adatokat is lehessen írni az egyes bejegyzésekbe, ezért így megoldható, hogy a rendszer bizonyos paramétereit futás közben módosíthassuk segítségével. Ezt a feladatot a `sysctl` (*system control* interfész rendszerhívások formájában tette lehetővé, ennek egy más struktúrájú kiegészítéseként hozták létre a `/proc/sys` jegyzéket, amelyel le lehet kérdezni, és be lehet állítani bizonyos rendszerparamétereket. Ebben olyan aljegyzékek szerepelhetnek, mint például a `debug`, `dev`, `fs`, `kernel`, `net` illetve `vm`.

Idővel egyre nagyobb szerephez jutott a proc fájlrendszer ilyen módú használata, amit így érdemesnek látszott külön egységként kezelni, és ez végül a `/sys` fájlrendszer létrejöttéhez vezetett. A `/proc/sys`-t természetesen még továbbra is használják, ezért ez a kettő, közel azonos funkcionalitással benne van a jelenlegi kernelekben. [8]

3.1. *Pszeudó-fájlrendszerek*

A pszeudó-fájlrendszerek olyan fájlrendszerek, amelyek csak a memóriában léteznek, eredendően nem foglalnak területet a háttértálókon. (Ez alól persze kivételt képez az az eset, amikor már a kilapozásukra is szükség van.)

Használatuknak számos előnye van, mivel

- egységes hozzáférést biztosítanak (`/dev` fájlrendszer, speciális eszközfájlok),
- rajtuk keresztül szabályozottan betekintést nyerhetünk a kernel bizonyos részeibe (`/proc` fájlrendszer),
- a kernel paraméterei módosíthatóak segítségükkel (`/sys` fájlrendszer),
- átmeneti fájlok tárolása így hatékonyabb megoldható (`/tmp` fájlrendszer).

3.2. A proc fájrendszer felépítése

A fájlrendszerben minden folyamathoz egy külön jegyzék tartozik. A jegyzék neve megegyezik a processz `PID`-jével. A jegyzéken belül az alábbi bejegyzések találhatók:

- `cmdline`: a folyamat elindításához használt parancsot tartalmazza,
- `cwd`: szimbólikus link az aktuális munkakönyvtárra (**c**urrent **w**orking **d**irectory),
- `environ`: a folyamatból látható környezeti változók listája,
- `exe`: szimbólikus link a folyamat elindításához használt bináris állományra (ha még létezik),
- `fd`: egy jegyzék, amelyben a folyamat által megnyitott fájlokra mutató szimbólikus linkek vannak,
- `fdinfo`: a megnyitott fájlokhoz tartozó pozíciók és jelzőbitek vannak benne,
- `root`: arra a gyökérjegyzékra mutat, amit a folyamat lát,
- `stat`: a futási állapotot írja le,
- `statm`: a memóriahasználatot foglalja össze,
- `status`: a futási állapot leírása olvashatóbb formában,
- `task`: egy jegyzék, amelyben az elindított gyerekprocesszekre mutató linkek vannak,
- `map`: a memória és a fájlok kiosztása érhető el benne.

Az egyes folyamatok jellemzőin kívül rendszerszintű információkat tartalmazó állományok is vannak a fájlrendszerben. A `cmdline` a kernel elindításakor átadott opciókat tartalmazza, a `cpuinfo`-val a processzor(ok) adatai jeleníthetők meg, az `uptime` a rendszer elindítása óta eltelt időt, a `swaps`-ból pedig a lapozófájlok adatai olvashatók ki. További lényeges információk is helyet kaptak még a fájlrendszerben, viszont ezek mindegyikére egyesével kitérni nem érdemes; a dolgozat bizonyos részein találhatók ezekre még további utalások.

A proc fájlrendszerhez tartozó részek a kernel forráskódjában az `fs/proc` jegyzékben találhatóak. Ebben bizonyos bejegyzések forrásállományait név szerint is megkereshetjük (pl.: `stat`, `vmstat`, `uptime`). A `/proc` bejegyzéshez tartozó beállítások a `root.c` fájlban szerepelnek.

A következőkben bemutatásra kerül, hogy mik azok a függvények, amelyekkel a rendszermagon belül módosítani tudjuk a fájlrendszert.

3.3. Új bejegyzés létrehozása

A proc fájlrendszerben a különböző típusú bejegyzések létrehozásához különböző függvények állnak rendelkezésre. Nyilvánvalóan ezeket a kernelből lehet csak meghívni. A függvények használatához a `linux/proc_fs.h` állományt kell hozzáadni a programunkhoz. A legegyszerűbb eset az, amikor egy új jegyzékkel szeretnénk bővíteni a fájlrendszert. Ehhez az alábbi függvény használható:

```
struct proc_dir_entry* proc_mkdir
(
    const char* name,
    struct proc_dir_entry* parent
);
```

A `name` paraméter itt az új jegyzék nevét, a `parent` pedig a szülő jegyzék nevét adja meg. Ha közvetlenül a proc fájlrendszer gyökerébe szeretnénk létrehozni, akkor a második paraméternek elég a `NULL` értéket megadni. A függvény a bejegyzés mutatójával tér vissza.

A reguláris fájlok létrehozása a következő függvénnyel történik:

```
struct proc_dir_entry* create_proc_entry
(
    const char* name,
    mode_t mode,
    struct proc_dir_entry* parent
);
```

Ebben már szerepel egy `mode` paraméter is, amivel a hozzáférési módot állíthatjuk be a hagyományos *rwx* (**r**ead - **w**rite - e**x**ecute) formátumban. Mint látható, mindkét függvény ugyanolyan típussal tér vissza. Ezzel a megoldással a különböző típusú fájlok teljesen egységesen kezelhetőek. A fájlrendszerbe létrehozhatók még szimbólikus linkek és speciális eszközfájlok is.

A fájlok eltávolítása a következő függvénnyel történik:

```
void remove_proc_entry
(
    const char* name,
    struct proc_dir_entry* parent
);
```

A paraméterek itt tulajdonképpen csak az útvonalat adják meg két részre bontva. A bejegyzéseket minden esetben el kell távolítani a fájlrendszerből, mert egy felesleges, esetleg rosszul beállított fájl biztonsági kockázatot jelent, vagy akár a teljes rendszer összeomlását okozhatja.

3.4. *Adatok átvitele*

A fájlrendszer egy átviteli közeget biztosít a kernel és a felhasználói tér között. Az adatok átvitelének módjaihoz először a `proc_dir_entry` struktúrát kell szemügyre venni. Ez a szokványos bejegyzésekhez tartozó információkon túl (pl.: név, hozzáférési jogok, tulajdonos) tartalmazza még az alábbi függvényekre mutató mezőket:

```
read_proc_t *read_proc;
write_proc_t *write_proc;
const struct file_operations *proc_fops;
```

Ezeknek a mezőknek értékként callback jellegű függvényeket lehet megadni.

A `read_proc_t` típusdefiníciója például az alábbi:

```
typedef int (read_proc_t)
(
        char *page,
        char **start,
        off_t off,
        int count,
        int *eof,
        void *data
);
```

A `page` itt annak a memóriaterületnek a címét mutatja, ahová majd a hívó program várni fogja a kiolvasott adatokat.

Az `off` paraméter a kiolvasandó adat kezdőcímét (*offset*-jét) adja meg. Az olvasás során maximum a `count` paraméterben megadott mennyiségű bájt kiolvasása lehetséges. Az `eof` címen megadott érték 1-re állításával jelezhetjük, hogy a fájl végére értünk. A `data` olyan esetekre van fenntartva, ha egy ilyen függvényt több fájlnál is szeretnénk használni, mivel a `proc_dir_entry` struktúrának van egy `data` mezője, amivel meg lehet adni a bejegyzésehez tartozó adatokat, majd az olvasásnál azt így visszakaphatjuk.

Egy másik hozzáférési mód is elképzelhető a `proc_fops` mező beállításával, viszont itt egy közvetettebb megvalósítás szükséges. A `file_operations` struktúrában benne van az összes lényeges fájlművelet. Ezekből most csak az alábbiakra lehet szükség

```
struct file_operations {
...
  ssize_t (*read)
    (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write)
    (struct file *, const char __user *, size_t, loff_t *);
  loff_t  (*llseek)
    (struct file *, loff_t, int);
...
};
```

Ez egy általánosabb megoldást ad a fájlok kezeléséhez, viszont mint a kevesebb számú paraméterből látható, az interfész által biztosított szolgáltatás is kevesebb. Ez esetben nem tudjuk jelezni a fájlvégeket, illetve a fájlok megkülönböztetése is külön megoldást igényel.

A későbbiekben szükséges lesz majd a fájlokba való írásra is. Ez, ahogy a függvények szignatúrája mutatja, teljesen megegyezik az olvasással, csak az adott bufferből olvasni lehet az adatokat, nem pedig beleírni (ezt jelzi a `const` megszorítás is).

A fájrendszerbe még létrehozhatók szimbólikus linkek illetve speciális eszközfájlok is. A szimbólikus linkeknek a folyamatok adatainak tárolása szempontjából ott van szerepe, hogy velük sokkal egyszerűbben megadhatók az útvonalak, mint ha azokat szöveges formában tárolnánk.

4. fejezet

A PROC FÁJLRENDSZERRE ÉPÜLŐ FELHASZNÁLÓI PROGRAMOK

A proc fájlrendszerhez használatához nem feltétlenül kellene külön programokat készíteni, mivel az állományok szöveges formában szolgáltatják az adatokat, és így egyszerűbb esetben elég megkeresni az információkat tartalmazó fájlokat és kiíratni a tartalmukat. Bonyolultabb lekérdezéseknél, összesítéseknél ez már nem járható út, ekkor már célszerűbb valamilyen felhasználói programot használni. A következőkben részekben olyan felhasználói programokról lesz szó, amelyek működéséhez elengedhetetlen a proc fájlrendszer megléte, és kényelmes felületet biztosítanak ahhoz, hogy adatokat kérdezzünk le abból.

4.1. *A `procps` csomag*

A Linux rendszerekben a programokat csomagokba szokás rendezni. Mivel a rendszer folyamatainak megfigyelése az általános feladatok közé sorolható, ezért ez egy külön *procps* nevű csomagba került, amely már gyakorlatilag az összes disztribúció részét képezi.

Érdemes megnézni a csomagból azokat a programokat, amelyek a processzek kezelésével közvetlen kapcsolatban vannak, illetve a memóriahasználatot jelenítik meg:

- `free`: A rendszerben használt fizikai és lapozáshoz használt memória mennyiségéről ad jelentés.
- `kill`: Jelzéseket lehet vele küldeni a folyamatoknak.
- `pgreg`: Folyamatokat lehet vele visszakeresni a nevük, vagy egyéb jellemzőjük alapján.
- `pkill`: Jelzés küldhető a névvel, vagy jellemzőjével megadott folyamatnak.
- `pmap`: Egy adott processz memóriakiosztásáról ad tájékoztatást.
- `ps`: Az aktuálisan futó folyamatokat lehet kilistázni vele.
- `pwdx`: A folyamat munkakönyvtárát adja vissza.
- `skill`: Az adott kritériumnak eleget tevő folyamatnak lehet vele jelzést küldeni.
- `snice`: A folyamatok ütemezését lehet vele befolyásolni.
- `top`: A legtöbb processzoridőt használó folyamatokat listázza ki.
- `vmstat`: A virtuális memória állapotáról állít össze egy statisztikát.
- `watch`: Egy adott folyamatot lehet vele periodikusan futtatni.

További programok is tartoznak még a csomaghoz, amelyek a rendszer leterheltségével, a gyorsítótárakkal, illetve a bejelentkezésekkel kapcsolatosak. A felsoroltak közül a dolgozat szempontjából a ps és top programok a lényegesebbek, mivel ezekkel lehet lekérdezni a folyamatok állapotát.

4.1.1. *Lekérdezések*

Mint az eddigiekből már kiderült a folyamatok állapotleírásához rengeteg információ tartozik hozzá. Egy lekérdezés során általában nem vagyunk kíváncsiak az összesre, csak egy részét szeretnénk visszakapni. A parancssori eszközök esetében ez úgy jelenik meg, hogy parancssori argumentumokként átadjuk a szűrési feltételeket, interaktív programoknál pedig futás közben is módosíthatjuk ezeket.

A rendszer monitorozása során érdekelhet bennünket, hogy összesen mennyi processz is van jelen. Ekkor csak az általános attribútumokra van szükségünk, mint például a folyamat azonosítójára, a tulajdonosra, a processzorhasználatra illetve arra, hogy milyen parancsot hajt végre a folyamat.

Szűrési feltételek megadásával rövidebb, egyszerűbben kezelhető kimenetet kaphatunk. Folyamatok esetében ez leginkább értékegyezőségre vonatkozó feltételeket jelenthet, vagy egy számértékhez adhatunk meg alsó- illetve felső küszöbértéket.

Az értékek sorbarendezése is segíthet minket abban, hogy a rendszerben fennálló aktuális állapotokat jobban megértsük.

A UNIX rendszerek esetében tervezési szempont, hogy egy adott problémát lehetőleg csak egyszer, de akkor minél jobban oldjunk meg. Ez az állapotstatisztikák esetében azt jelenti, hogy mivel szöveges kimeneteket kapunk, azt már a tetszőleges parancssori alkalmazással (pl.: sed, awk, sort) könnyedén elvégezhetjük. Ha még ezek sem lennének elegendőek a kívánt statisztika összeállításához, akkor shell-szkripteket is írhatunk segítségükkel.

4.1.2. *A ps program*

A ps felhasználói program szolgál a processz állapotok lekérdezésére. Megtalálható minden UNIX rendszeren, így a Linux disztribúcióknak is szerves részét képezi. Ebben az alfejezeteben a ps program használata, szerkezete és működése kerül bemutatásra.

Mindhárom elterjedt kapcsoló megadási módot is támogatja, úgy mint

- a hagyományos, UNIX-ban használt egy kötőjeles kapcsolókat (pl.: `ps -V`),
- a BSD-kben használt csoportosított, kötőjel nélküli kapcsolókat (pl.: `ps V`),
- a GNU hosszú-, két kötőjellel megadott kapcsolókat (pl.: `ps ---version`).

Ezeket a módokat egyidejűleg is lehet használni, ami sajnos félreértésekre adhat okot.

A listázandó folyamatok kiválasztásának két fő módja van. Az első esetben általánosan adhatjuk meg, hogy mely folyamatok adataira vagyunk kíváncsiak. Az összes folyamat listázásához a `ps -A`, illetve az ezzel megegyező `ps -e` parancsok használhatók. Az adott terminálhoz tartozó folyamatokat a `ps t` vagy `ps T` parancsokkal, az éppen futó folyamatot pedig a `ps r` parancsal kaphatjuk meg.

A másik esetben egy lista alapján végzi el a program a szűrést. Ez lehet például a folyamatazonosítók listája (pl.: `ps ---pid 11,12,16`), a felhasználók neve vagy azonosítója (pl.: `ps -u 33,81`) illetve a kiadott parancs neve is (pl.: `ps -C udevd,httpd`).

Amennyiben nem adunk meg a szűréshez paramétereket, akkor az aktuális effektív felhasználói azonosítóhoz, és terminálhoz rendelt processzek lesznek kiválasztva.

A kimenet formázásához meg kell adni, hogy mely mezők szerepeljenek az eredményben. A mezőket különféle szempontok szerint csoportosították, így azokat nem szükséges minden

esetben külön megadni. Alapértelmezés szerint a program a folyamat azonosítóját (PID), a terminál nevét (TTY), az összesített CPU használatot (TIME) illetve a kiadott parancsot (CMD) írja ki.

Számos előre definiált lekérdezéstípus van, így például lekérdezhetők a regiszterekkel (ps X), jelzésekkel (ps s), virtuális memóriával (ps v) illetve biztonsági információkkal (ps Z) kapcsolatos adatok is.

Lehetőség van a mezők explicit megadására, méghozzá a -o kapcsolóval. Ez után a mezők listájának kell szerepelnie. A mezők több néven is adottak, például a ps -o user,comm és a ps -o uname,ucmd ugyanazt az eredményt adják (a mezőnévben van csak eltérés).

A kimenet előállításánál a kiválasztás a paraméterek tekintetében nem különül el teljesen, például a ps u parancs önmagában már a felhasználó szempontjából fontosabbnak vélt folyamatokat és mezőket is meghatározza.

A kimeneten további módosítások is lehetségesek, például rendezhetjuk valamelyik mező értéke alapján (pl.: ps ---sort cmd).

További lehetőség még, hogy mivel a folyamatok hierarchiában helyezkednek el, ezért az eredményt is ilyen formában várjuk. A ---forest kapcsoló hatására a program fa struktúrában jeleníti meg a folyamatok adatait. (Egy másik lehetőség a struktúra megjelenítésére a pstree parancs használata.)

Az itt említett szintaktikai elemeken túl még rengeteg egyéb megadási módot is támogat a program. Ez leginkább annak köszönhető, hogy a fejlesztői igyekesztek a különböző UNIX rendszerek elvárásainak megfelelni, illetve biztosítani azt, hogy a régebben megírt szkriptekben a ma már nem javasolt megadási módok is működőképesek legyenek.

A program felépítése szempontjából két fő funkciót különböztethetünk meg; egyrészt a processzek adatait össze kell gyűjtenie, másrészt a kapott argumentumok alapján meg kell formáznia a kimenetet. A hatékony lekérdezéshez mindkettőre szükség van.

A forráskódban a processzek adatainak kezeléséhez szükséges dolgok egy proc jegyzékben kaptak helyet. A csomagban található programok ezt használják arra, hogy hozzá tudjanak férni a proc fájlrendszerhez. Az ebben lévő procps.h-ban van egy saját folyamat leíró struktúra definíciója. Ez jóval rövidebb, mint a sched.h-ban található, és törekszik az adatokat a lehető legegyszerűbb formában tárolni. Ebben már nem a kernelben lévő típusdefiníciók szerepelnek, hanem a mezők közvetlenül a megfelelő C típussal adottak. Ennek a struktúrának a mérete már csak 536 bájt.

A másik alapvető struktúrája a PROCTAB. Ennek procfs mezője magára a proc fájlrendszerre mutat. Van benne négy függvénymutató (finder, reader, taskfinder, taskreader), amellyel majd be tudja járni, és olvasni a folyamatok és szálak adatait. A struktúra inicializálása a readproctab függvényhívással kezdődik, amely a kapott jelzőbiteknek és argumentumoknak megfelelően az openproc függvénnyel fogja a mezőket beállítani. Ez a függvénymutatókhoz a következő függvényeket rendeli:

finder	→	simple_nextpid
reader	→	simple_readproc
taskfinder	→	simple_nexttid
taskreader	→	simple_readtask

Az így beállított

```
proc_t * simple_nextpid(
        PROCTAB *restrict const PT,
         proc_t *restrict const p
);
```

fog majd végigiterálni a proc jegyzékben, és a beolvasott, csak számjegyeket tartalmazó (tehát folyamathoz tartozó bejegyzések) útvonalát beállítja a PROCTAB->path mezőben. Ezután az adatok kiolvasását a

```
proc_t * simple_readproc(
        PROCTAB *restrict const PT,
         proc_t *restrict const p
);
```

függvény végzi el. Ez a következő műveleteket hajtja végre:

- Beállítja a proc bejegyzés útvonalát, és a jelzőbiteket a PT alapján,
- stat függvényhívással beolvassa a fájl attribútumait, majd ezekből beállítja a felhasználói és csoportazonosítókat,
- sorban beolvassa a fájlokat, és meghívja a megfelelő függvényt.

  stat → stat2proc

  statm → statm2proc

  status → status2proc

  A cmdline és environ esetében csak értékátadás történik.

Az egyes folyamatokhoz tartozó bejegyzéseken belüli task jegyzék feldolgozása a simple_nexttid és simple_readtask függvényekkel teljesen analóg módon működik.

A program működését, a forráskód tanulmányozása nélkül is vizsgálhatjuk az strace nevű programmal. Ennek paraméterként meg kell egy parancsot, majd futás közben vizsgálhatjuk, hogy az milyen rendszerhívásokat hajtott végre. A -c kapcsolóval a futást követően egy összesítést is kaphatunk arról, hogy melyik rendszerhívást hányszor hívta meg, illetve hogy az a futási idő hány százalékát tette ki.

A strace -c ps ax parancs az alábbi kimenetet adja 109 folyamat esetén:

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000019           0       340         1 open
  0.00    0.000000           0       350           read
  0.00    0.000000           0       110           write
  0.00    0.000000           0       337           close
```

Mivel kis számú folyamat esetén, minimális terheltségű rendszeren ennek a futási ideje elhanyagolható, ezért az open() rendszerhívásnál szereplő százalékértékkel most nem kell különösebben foglalkozni, ez futásonként változik attól függően, hogy a programnak éppen mennyire pontosan sikerül lemérni az adott időintervallumot.

Mint látható a futás során több mint háromszor annyi open(), read() és close() rendszerhívást kellett végrehajtania a programnak, mint amennyi a folyamatok száma. (A folyamatok száma egyszerűen meghatározható, mivel a ps a kimeneten egy fejlécet jelenít meg, illetve soronként a folyamatok adatait, így az egyel kevesebb, mint a write() rendszerhívások száma.) A nagy számú rendszerhívás annak köszönhető, hogy minden folyamat esetén szükség van a stat, statm illetve status fájlok beolvasására.

4.1.3. *A top program*

Az operációs rendszer működésének nyomonkövetéséhez adott időközönként meg kell ismételni a lekérdezéseket. A top egy interaktív program, ami elsősorban ezt a feladatot látja

el. A program kimenete két fő részből épül fel; a felső részen a teljes rendszerre vonatkozó összesített adatok találhatóak, alatta pedig a folyamatok listája valamelyik mező szerint rendezve.

A megjelenítés mellett lehetőség van jelzések küldésére is a folyamatoknak a `k` billentyűvel (*kill*), vagy módosíthatjuk a prioritásukat az `r`-rel (*renice*).

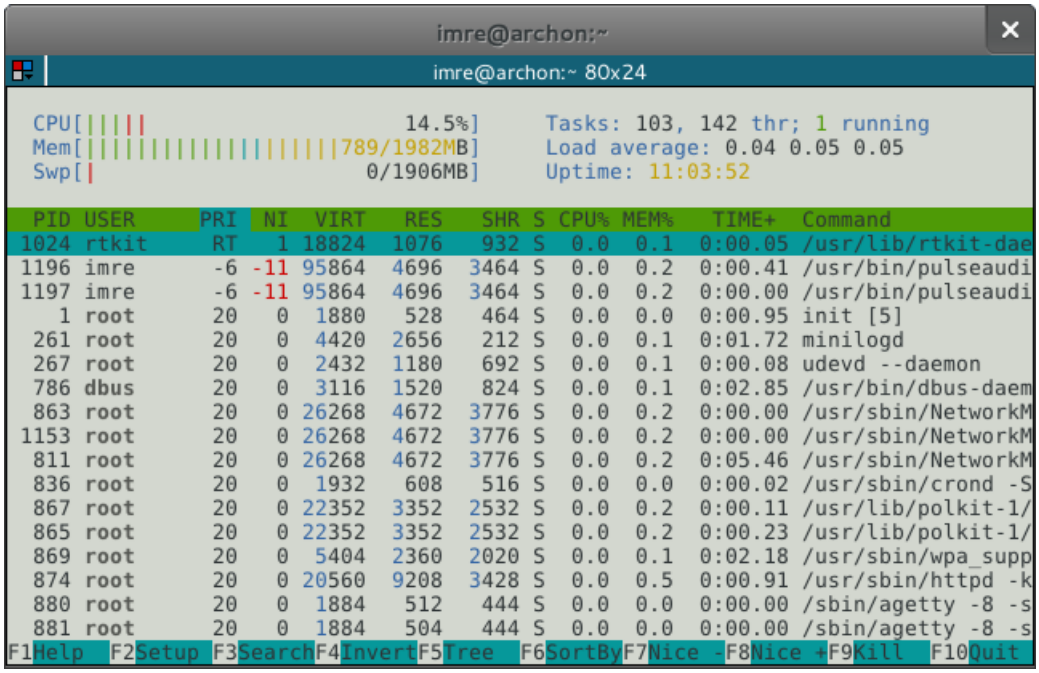
A program nagyon rugalmasan kezelhető:

- az összegző részben lévő sorok megjelenítését külön állíthatjuk,
- kiválaszthatjuk, hogy az eredménylistában milyen mezők szerepeljenek, és melyik mező szerint legyen rendezve,
- megadhatjuk a frissítési intervallumot másodpercben,
- adhatunk egy felső limitet a megjelenített folyamatok számára,
- beállíthatunk felosztott nézetet különböző mezőcsoportokkal,
- beállíthatjuk a megjelenítés szineit.

4.2. *A `htop` program*

A `htop` a `top` program egy továbbfejlesztett változata. Ez már *NCurses*-t használ a megjelenítéshez (4.1 ábra). Az adatokat a `top` programéhoz hasonló elrendezésben szolgáltatja, viszont itt ebben már lehetőség van a listák görgetésére (függőlegesen és vízszintesen is), és nem szükséges begépelni a folyamat azonosítóját, mivel ki is lehet már választani a listából.

A megjelenítés és kezelés szegényesnek tünhet a manapság asztali környezetben használt grafikus programokéhoz képest, viszont azt is figyelembe kell venni, hogy a szerverek működéséhez nem feltétlenül kell grafikus környezetet telepíteni, így ha a monitorozást végző felhasználói program az adott rendszeren fut, akkor ilyen lehetőségekben gondolkodhatunk.



4.1. ábra. A `htop` program NCurses alapú felhasználói felülete

4.3. *Az `lsof` és `fuser` programok*

Eddig olyan esetekről volt szó, amelyekben a folyamat általános jellemzői alapján történt a lekérdezés, majd azok eredményeit szűkítve, esetleg rendezve adta vissza a program a végeredményt. Léteznek azonban speciális lekérdezések is, amelyeknél például arra lehetünk kíváncsiak, hogy egy adott fájlt (ami jelenthet egy eszközt is) melyik folyamat használja. A keresett információ ekkor is a processztáblában található, méghozzá az `fd` jegyzékben lévő szimbólikus linkek formájában.

Az `lsof` és az `fuser` program is arra szolgál, hogy a megnyitott fájlokról adjon vissza információkat. Az `lsof` (LiSt Open Files) program az aktuálisan megnyitott fájlokat listázza ki, ha nem kap külön paramétereket. Kimenetén megjelenő fontosabb jellemzők:

- `COMMAND`: A fájl megnyitásához használt parancs első 9 karakterét jeleníti meg.
- `PID`: A fájlt használó processz azonosítója.
- `USER`: A folyamat tulajdonosa.
- `FD`: Fájl leíró, amely egy számérték, vagy egy jellemző (pl.: `rtd` - gyökér jegyzék). Számérték esetén ezt követi a megnyitás módja (`r`/`w`/`u`, ahol `u` írható és olvasható is), illetve még zárolással kapcsolatos információk is.
- `TYPE`: A fájl típusa (pl.: `REG`, `CHR`, `FIFO`).
- `DEVICE`: Az eszközök azonosító száma vesszővel elválasztva.
- `SIZE/OFF`: A fájl mérete (ha elérhető), illetve az aktuális kezdőcím.
- `NODE`: A bejegyzés azonosító száma.
- `NAME`: A megnyitott fájl neve.

Paraméterként egy fájl nevét megadva megkaphatjuk, hogy az adott állományon éppen mely folyamatok dolgoznak. Természetesen ennek a programnak is rengeteg egyéb paraméterezési lehetősége van.

Az `fuser` (File USER) program az `lsof` egy célirányosabb változatának tekinthető. Alapértelmezés szerint csak a fájlt használó folyamatok azonosítóját adja vissza. A programmal jelzéseket is küldhetünk a `-k` kapcsoló segítségével, így könnyebben leállíthatjuk az azt használó folyamatokat, mint ha `lsof` és `kill` parancsok kombinációjával oldanánk meg.

A működésüket illetően mindkét programban a `/proc` bejegyzéseinek végigolvasásáról van szó. A programok hatékonyságán így a processztáblához való gyorsabb hozzáféréssel szintén segíteni lehetne, viszont ezekre a programokra ritkábban van szükség, és akkor is csak egy-egy lekérdezés erejéig, nem úgy mint például a `top` esetében.

4.4. *Lekérdezés hálózaton keresztül*

Amikor hálózaton keresztül távolról kell monitorozni, vagy menedzselni egy rendszert, akkor gyakran alkalmazzák az SNMP (**S**imple **N**etwork **M**anagement **P**rotocol) protokollt. Már az 1980-as évek közepe felé felmerült az igény, egy egyszerűen használható, rugalmas módszerre, mivel a hálózatok nagyon gyorsan kezdtek terjeszkedni és felügyeletüket meg kellett oldani valahogy. Az első ajánlás az RFC 1067 -es volt 1988-ban. Ebben megtalálható az SNMP használatához szükséges architektúra, az információ kezelésének módja és a protokoll által támogatott műveletek is. Ezt később másik kettő is követte, méghozzá az RFC 1098

és RFC 1157, de ezek csak minimális korrekciókat tartalmaznak, maga a protokoll nem változott. A későbbi bővítésekhez további RFC-k is készültek (RFC 1902-1907), amelyek az átadott információ struktúráját (**M**anagement **I**nformation **B**ase), a műveleteket illetve az átviteli módokat taglalják.// Ref az RFC-kre

A protokollt főként hálózati eszközök megfigyelésére fejlesztették ki, de használata nem korlátozódik kizárólagosan csak azokra, mivel tetszőleges információ lekérdezhető a segítségével. Kiszolgálók üzemeltetésénél gyakori, hogy magáról az operációs rendszerről kérdeznek le információkat. Ezek között szerepelhetnek az aktuálisan futó folyamatok is, így megoldható vele azok távoli megfigyelése is.

A protokoll kliens-szerver arhitektúrára épül. A megfigyelt eszközön fut az SNMP kiszolgáló (*daemon*), amely a megfigyelést végző programtól kap kéréseket, amelyre válaszként küldi vissza az igényelt információt.

Az operációs rendszerek számára egy elterjedt implementációja a `net-snmp` csomagban található. Ebben van benne a kiszolgáló program is, ami hozzá tud férni a rendszerben futó folyamatok adataihoz is. Számos operációs rendszert támogat, beleértve a nem UNIX-szerű operációs rendszereket is.

A Linux rendszerekhez készült implementációban ennél is a proc fájlrendszer szolgáltatja a rendszer működésével kapcsolatos információkat. A `/proc`-ban lévő bejegyzések útvonalaihoz a következő definíciókat használja:

```
#define CPU_FILE      "/proc/cpuinfo"
#define STAT_FILE     "/proc/stat"
#define VMSTAT_FILE   "/proc/vmstat"
#define MEMINFO_FILE  "/proc/meminfo"
```

A `/proc` jegyzésben lévő azonosító szerinti bejegyzések végigolvasásához a `swrun_procfs_status.c` forrásfájlban lévő `netsnmp_arch_swrun_container_load` függvényt használja. Ez végigjárja a folyamatokhoz tartozó bejegyzéseket, és feldolgozza az azokban lévő `status`, `cmdline` illetve `stat` bejegyzéseket. A feldolgozás úgy történik, hogy a benne lévő adatokat összerakja egy `netsnmp_swrun_entry` struktúrába, majd ezeket beszúrja egy `netsnmp_container`-be.

A processzek adataihoz tehát ez a program is a proc fájlrendszeren keresztül jut, minden folyamat esetén külön megnyitva a hozzá tartozó fontos leíró fájlokat, amiket azt követően, mint szöveges fájlokat kezel.

5. fejezet

A KERNEL BŐVÍTÉSI LEHETŐSÉGEI

Mivel a feladat megoldásához rendszerszintű műveletekre van szükség, ezért meg kell nézni, hogy ezeket hogyan is lehet egyszerűen integrálni a jelenleg is működő rendszerbe. Az egyik kézenfekvő megközelítés az lehetne, hogy a bővítéseket magában a rendszermag forráskódjában végezzük el. Ez különösebb problémát abból a szempontból nem okoz, hogy a teljes kód szabadon hozzáférhető, és a lefordítása során sem ütközünk komolyabb akadályokba. Számos ellenérv szól viszont ellene, ami miatt más megoldást kell keresni.

Az egyik ezek közül, hogy a kernel lefordítása egy hosszadalmas folyamat, és az új kernel telepítése is körülményes lehet. A program tesztelése és karbantartása nehézkes; a változtatások után a teljes rendszermagot le kell cserélni, és tesztelés közben a rendszer újraindítására szükség lehet.

5.1. *Kernel modulok*

A Linux egy monolitikus kernel, tehát a rendszer működéséhez szükséges feladatok mind privilegizált módban futnak. Ez azt eredményezi, hogy így azokhoz a hozzáférés nem annyira egyszerűen oldható meg, mint a mikrokernelek esetében, ahol ezek egy része már felhasználói módban fut. Ezt a problémát oldotta meg a kernelbe dinamikusan betölthető kernel modulok bevezetése.

A kernel modulok nem voltak benne az első Linux kernelekben, csak az 1.2 - es verzióval (1995) kerültek bele. Ez a bővítés jelentősen leegyszerűsítette többek között az eszközmeghajtók fejlesztését is. A modularitásnak több, főként előnyös tulajdonsága van:

- A modulok nem foglalnak külön helyet a kernelben, ha nincs rájuk szükség.
- Egyszerűbben lehet tesztelni és karban tartani. Hiba esetén a problémás rész is könnyebben visszakereshető, mint ha az közvetlenül a kernelben lenne.
- A kernelmodulokban lévő kód ugyanolyan gyorsan fut le, mint ha azok a kernelel együtt lettek volna lefordítva és statikusan linkelve hozzá. Ami plusz időt igényel, az mindössze a modul betöltése.

A kernel modul valójában egy speciális futtatható és linkelhető (ELF - Executable and Linkable File) fájl. Mivel a szimbólumok feloldása a modul betöltése előtt nem lehetséges, ezért csak, mint object fájl fordítható le.

A modulok kezeléséhez a `modutils` csomag biztosít eszközöket. A modulok betöltését a rendszerbe a `insmod`, eltávolításukat pedig az `rmmod` parancsokkal tehetjük meg. Mivel a modulok között függőségi viszonyok lehetnek, ezért általában inkább a `modprobe` parancsot

érdemes használni, ami automatikusan betölti a helyes működéshez szükséges további modulokat. Az aktuálisan betöltött modulok listázásához az `lsmod` használható, amely a proc fájlrendszerben lévő `modules` bejegyzésből dolgozik.

A létrehozásuk nagyon egyszerű. Az alábbi vázlatos C program áttekintést ad a felépítésükről:

```
#include <linux/kernel.h>
#include <linux/module.h>
...
MODULE_LICENSE("GPL");
...
static int __init sample_module_init(void) {
    ...
}
...
static void __exit sample_module_exit(void) {
    ...
}

module_init(sample_module_init);
module_exit(sample_module_exit);
```

A `kernel.h` és `module.h` fájlok megadása szükséges ahhoz, hogy a kód kernelmodulként forduljon le, és el tudjuk érni az alapvető függvényeket.

A modul adatait beállíthatjuk a `module.h`-ban lévő makrók segítségével, mint például a `MODULE_LICENSE`, `MODULE_AUTHOR` vagy `MODULE_DESCRIPTION`. A licensz megadása a mintában opcionálisnak tünhet, de nem az. Bizonyos szimbólumokat, csak akkor használhatunk, ha a modul GPL licenszel vannak ellátva.

A `sample_module_init` függvény inicializálja majd a modult, a `sample_module_exit` pedig elvégzi az eltávolítás előtt szükséges műveleteket. A `module_init` és `module_exit` makrók arra szolgálnak, hogy a modulban ezeket beállítsák, mint kapcsolódási pontokat.

Az `__init` és `__exit` az `init.h`-ban definiáltak. Ezek jelzik a kernelnek, hogy az adott függvényekre mikor lesz szükség, és így az ennek megfelelően fogja majd a helyet felszabadítani.

Bár a modulok a kernelbe töltődnek be, és így tulajdonképpen annak a kódját egészítik ki, mégsem érhető el belőle a kernel minden része. A dinamikus linkelés miatt csak azokat a szimbólumokat érhetjük el, amelyek ki vannak exportálva. A `printk()` függvényt például a

```
EXPORT_SYMBOL(printk);
```

makróval adták meg, hogy a betöltött modulból használható legyen. Az exportálásnak a másik módja az `EXPORT_SYMBOL_GPL` makróval lehetséges, ahol azt adjuk meg, hogy csak a GPL licenszel ellátott modulok használhassák a szimbólumot. A bővítés szempontjából azért kell figyelnünk ezekre a dolgokra, mert ha egy, a kernel kódjában lévő függvényt a modulban szeretnénk használni, és az nem lenne exportálva, akkor annak implementálása körülményes lehet, az exportáláshoz pedig újra kell fordítani a kernelt, így mérlegelnünk kell, hogy melyik is lenne a jobb megoldás.

6. fejezet

AZ ÚJ LEKÉRDEZÉSI MÓDSZER

Annak érdekében, hogy a lekérdezés gyorsan lefuthasson az adatokat minél nagyobb blokkokban érdemes mozgatni. A lekérdezést így az alábbi fő lépésekre lehet bontani:

- A kernelt értesíteni kell arról, hogy adatokat szeretnénk lekérdezni.
- A kernelben össze kell gyűjteni az adatokat, majd egy meghatározott struktúrába rendezni.
- Az adatokat át kell juttatni a felhasználói térbe.
- A felhasználói programnak fel kell dolgozni a kapott struktúrát; formázni esetleg rendezni kell, majd megjeleníteni.

A jelenleg használt architektúra úgy épül fel, hogy a kernel láthatóvá teszi a processzekkel kapcsolatos információkat a fájlrendszer bizonyos pontjain, majd a felhasználói program azokat külön összegyűjti, rendezi és megjeleníti.

A lekérdezésnél csak kompromisszumos megoldások képzelhetők el. Az a program, amelyik a monitorozást végzi, maga is egy folyamat, tehát más folyamatok a futását félbeszakíthatják, így lekérdezés közben a lista is változhat. Ez kiküszöbölhető lenne, viszont akkor az összes többi programnak meg kellene várni, hogy a lista elkészítése befejeződjön, és ezzel a monitorozó program túlságosan nagy prioritást kapna. A `ps` program esetében is csak annyi történik, hogy sorban végigmegy az egyes folyamatokon, majd visszaadja az eredményt. Ha lekérdezés közben egy folyamat leállna, vagy létrejönne akkor az a kimeneten attól függően jelenne meg, hogy az éppen vizsgált processz előtt, vagy után volt. Így az, hogy a listát csak egyszer olvassuk végig, és közben nem figyelünk az esetleges változásokra ésszerű komprimusszumnak tekinthető.

Nagyobb mennyiségű bináris adat átvitelére jó példát ad a `config.gz` fájl (Ez a kernel aktuális beállításait tartalmazza; a kernel újrafordításánál lehet például hasznos). Ez a már bemutatott `file_operations` struktúra `read` függvényét használja az adatok átadásához. A `read_proc` függvény is megfelelő lenne, viszont ez képes nagyobb egységekkel is dolgozni. Ami később nehézséget okozhat, hogy ez viszont nem támogatja azt, hogy több bejegyzéshez is hozzárendeljük ugyanazt a függvényt.

A hatékonyságon az is javít, ha a szűréseket a lehető leghamarabb, már a kernel oldalon el tudjuk végzeni. Az átvitt adatok mennyisége így csökken, és a kernel oldalon az adatok könnyebben rendelkezésre állnak. Amit ebben az esetben meg kell oldani, hogy a szűrési feltételeket átjuttassuk a kernelnek.

Hosszabb listák, vagy gyakori lekérdezések esetén célszerű volna, ha a kernel tárolná azt, hogy hol tart a lekérdezés. Ezzel az a gond, hogy a kernelben ilyenkor nyilván kell tartani minden kéréshez a kérés adatait, egy pufferben tárolni, hogy milyen adatok következnek

majd, figyelni kell, hogy az igényelt adatok többi részére egyáltalán szükség lesz-e még, vagy esetleg a kérést kezdeményező folyamat már nem is létezik. A szélsőséges, konkurrens esetek vizsgálata nagyon fontos, mivel nem indulhatunk abból ki, hogy minden helyesen, rendeltetésszerűen fog mindig működni, a hibákat is kezelni kell.

A rendezés már nagyobb problémát jelent, ugyanis ahhoz a teljes lista szükséges. A kernelben ideiglenesen tárolni a lekérdezés adatait nem célszerű, ezért azt mindenképpen csak a felhasználói programban tehetjük meg.

A következőkben az egyszerűbb esetektől a bonyolultabbak felé haladva nézhetjük meg, hogy hogyan is valósul ez meg a gyakorlatban, és közben milyen további problémákkal kell szembenézni. Mivel a kernel- és felhasználói tér között kell kommunikálni, ezért minden esetben egy kernel modult, illetve egy ahhoz tartozó felhasználói programot kell vizsgálni.

6.1. *Folyamat azonosítók listázása*

Legegyszerűbb esetben elegendő csupán a folyamatok azonosítóját lekérdezni. A struktúra ekkor csak egy számértékre, a `PID`-re korlátozódik. Az adatokat bináris formában szeretnénk kezelni, tehát minden ilyen azonosító (az egyszerűség kedvéért most rögzítetten) 4 bájt hosszúságú lesz. A fájlok olvasását a C programozási nyelvhez készített szabványos függvénykönyvtárban lévő `fread()` függvénnyel végezhetjük. A függvény paraméterezése a következő:

```
size_t fread(
        void *buffer, size_t size, size_t count, FILE *file);
```

A `buffer` arra a tömbre mutat, amibe majd az adatokat be szeretnénk olvasni. A `size` az adategységek méretét, a `count` pedig az adategységek számát adja meg. A `file` a megnyitott fájlhoz tartozó struktúrára mutató pointer. Az adategységek mérete és száma leginkább csak a függvényt használó programnak szól, mivel az `fread()` függvénynek a további műveletekhez csak azt szükséges tudni, hogy mekkora az összes adat mennyisége.

A proc fájlrendszerben a bejegyzéseknek általában nincs rögzített méretük, mivel tartalmukat csak az olvasásuk során állítja össze a rendszer. Egy teljes fájl beolvasásához tudni kellene annak a maximális méretét, ami viszont így nem lehetséges. Az egyik megoldás az lehet, ha előbb lekérdezzük, hogy a processztábla mekkora területet foglal, majd annak foglalunk le előre memóriaterületet, és oda olvassuk be a fájl tartalmát. Ennél a megoldásnál sajnos további gondok jelentkezhetnek:

- A processztábla a méretének lekérdezése és az adatok kiolvasása között is változhat. Megtehetnénk, hogy valamennyivel több memóriát foglalunk le, mint amennyi a lekérdezésből kapott táblaméret, viszont nagyságrendet itt is nehéz mondani.
- A processzek adatai között vannak olyanok, amelyek mérete változhat, emiatt a méret megállapításához egyszer végig kell olvasni az egész táblát, ami így már közel megfelel az adatok lekérdezésének számításigényével.

A másik megoldás az, ha az adatokat több kisebb részben adjuk át. Ennél is több megoldandó feladat van, viszont ezek már könnyebben kezelhetőek.

Először is el kell dönteni, hogy mi legyen az átviteli egység, tehát egy lépésben milyen adatok átvitelére kerüljön sor. Kézenfekvő lenne, hogy egy egységben egy folyamat adatait juttassuk át, mert így a rendszerhívások száma közel harmada lesz annak, mint ami a jelenleg használt átviteli módszereknél van, viszont elképzelhető még ennél is hatékonyabb megoldás. Az adatok másolása a kernel és a felhasználói programok között adott méretű lapokkal történik. Az átvitelt úgy gyorsíthatjuk, ha minél nagyobb lapokat használunk, és igyekszünk

az azokon lévő helyet a lehető legjobban kihasználni. A jelenleg vizsgált egyszerű példában ez azt jelenti, hogy a `PID`-ekkel teljesen feltöltjük azokat.

Amennyiben az átvitel darabokban történik, valahogy nyilván kell tartani, hogy milyen adatok átvitele történt már meg, és melyeket kell még továbbítani. A kernel oldalon, mint az előbbi megoldás vizsgálatánál kiderült, nem érdemes nyomon követnie, hogy a lekérdezésben mi is szerepel. Egyfajta állapotmentességre kell törekedni. Ennek a lekérdezés közben esetleg változó processztáblák, és a konkurrens hozzáférés esetén van szerepe, mivel így az eredményeket szolgáltató modulnak mindig csak az aktuálisan lekérdezett adatokkal kell törődnie; nem kell tudnia, hogy azt éppen ki kérdezte le, és azt sem, hogy a lekérdezés éppen hol tart.

A legkisebb kezelhető egységnek mindenképpen egy processz leíróját kell választani. Ennél kisebb egység választása csak bonyodalmakhoz vezetne, illetve jelentős javulást sem eredményezne a teljesítményben. A folyamat azonosítója kiválóan alkalmas arra, hogy megadjuk vele, hogy honnan szeretnénk a lekérdezést folytatni. A modulnak ha átadjuk az utolsó folyamat azonosítóját, amelynek az adatait már megkaptuk, akkor abból már egyértelműen kiderül, hogy mi szerepeljen a következő lekérdezésben.

Az érték átadása sajnos nem triviális. Tulajdonképpen egy fájl beolvasásáról van szó, ezért az átadott `PID`-et a kezdőcím (*offset*) beállításával határozhatnánk meg, viszont nagyobb mennyiségű adat átadásánál a kernel, hogy hatékonyabb legyen a másolás a címeket lapméretre igazítja, így egy fájl beolvasását több lépésben oldhatja meg. Hagyományos fájlok esetében ez nem jelent gondot, mivel az adatok a fájl különböző részein rendelkezésre állnak, viszont a proc fájlrendszer esetében már igen, mivel az adatokat a kapott paraméterek függvényében szeretnénk előállítani.

A `file_operations` struktúrában beállított `read` függvény `len` és `offset` paraméterének értékeit kell megnézni, hogy hogyan változnak az `fread()` függvény `size` és `count` értékeinek a hatására, illetve hogy ezt hogy befolyásolja, ha az `fseek()` függvénnyel átállítjuk a kezdőcímet.

Az `len` paraméternek mindenképpen a kiválasztott lapméretnek kell lennie, ezért a `size` és `count` értékeket úgy kell megválasztani, hogy a szorzatuk azt adja. A kezdőcím, ha nem a lapok határára esik, akkor az olvasásnál ez úgy jelenik meg, hogy lesz egy kisebb méretű beolvasás a következő lap méretéig, majd az így kapott kezdőcímtől még egy a már megadott lapmérettel. Ezek alapján az `fseek()` függvényben beállított valódi `offset` értéket (későbbiekben `real_offset`) úgy kaphatjuk meg, ha az első beolvasásnál kapott `len` paramétert, illetve a másodiknál kapott `off` paramétert összeadjuk.

Ebből a szempontból már speciális helyzetnek tekinthető, ha a kezdőcím eleve laphatáron van, mivel így csak egy beolvasás történik. Ezek alapján a `real_offset` kiszámítása a `read()` függvényben az alábbiak szerint néz ki:

```
static ssize_t read(struct file *file,
    char __user *buf, size_t len, loff_t* offset)
{
        static loff_t real_offset = INVALID_OFFSET;
        int n;

        if (len < QUERY_PAGE_SIZE) {
                real_offset = *offset + len;
        }
        else {
                if (real_offset == INVALID_OFFSET) {
                        real_offset = *offset;
                }
```

```
                query(page, real_offset);

                real_offset = INVALID_OFFSET;
        }

        return len;
}
```

A `QUERY_PAGE_SIZE` és az `INVALID_OFFSET` saját definíció; az előbbi az egy lekérdezéshez tartozó lap méretét adja, míg az utóbbival azt jelezhetjük, hogy a kezdőcím nem érvényes. A `query` függvénybe kerülhet maga a lekérdezés, mivel az már pontosan megkapja, hogy milyen értéket adtunk át.

A lekérdezés során a darabok száma nem előre meghatározott, így jelezni kell, hogy melyik volt az utolsó érvényes leíró. Ehhez egy olyan értéket kell megadni, ami megkülönböztethető egy folyamat azonosítójától. Ez lehet például a 0 érték, de mivel a POSIX szabvány szerint (`posix_types.h`) a `pid_t` típusa előjelesként definiált, ezért tetszőleges negatív szám is lehetne. A későbbiekben a 0 értéknek egyéb speciális jelentése lesz, ezért az azonosítók listájának a végét a -1 érték zárja. Az ez után következő részt már nem használja a program.

Egy rövid lekérdezésre mutat példát a 6.1 táblázat. Ennél a lapméret 16 bájt (feltételezzük, hogy most ez is lapméret), és a processztáblában összesen 9 folyamat van, és a lekérdezés közben nem változik a tábla. A 0 kezdőcímmel azt jelezzük, hogy ez egy új lekérdezés.

offset		visszaadott tábla
0	$\rightarrow$	{1, 2, 4, 5}
5	$\rightarrow$	{7, 8, 11, 12}
12	$\rightarrow$	{15, -1, *, *}

6.1. táblázat. Az azonosítók listájának lekérdezése (ideális esetben)

A modulban vissza kell keresni az utoljára lekérdezett folyamat struktúrájának mutatóját, majd a processztáblában addig haladni előre, amíg még van hely az adatok feltöltésére.

Előfordulhat, hogy két lap lekérdezése között az utoljára átadott folyamat már befejeződött, így az az azonosító már nem érvényes. Ennek a vizsgálatát is el kell végezni, és ha már nem létezne, akkor azt jelezni kell a felhasználói program számára. Ezt úgy úgy tehetjük meg, hogy az átadott lap a 0 értékkel kezdődik. Szebb megoldás lenne, ha azt tudnánk visszajelezni, hogy a fájl adott pozícióban nem olvasható, viszont a kezdőcím kiszámítása miatt ez nem lehetséges. A felhasználói programnak ezután újra kell próbálkoznia egy korábbi `PID` értékkel, illetve törölnie az utolsó folyamatot a listából, mivel az már nem létezik. Egy ilyen esetre mutat példát a 6.1 táblázat.

6.2. *Adatstruktúra átvitele*

Az adatstruktúra átvitelez az előző részben bemutatott átviteli módból már következik. Annyi különbség van, hogy ekkor az egyes mezők típusára és méretére is figyelmet kell fordítani. A lapok kihasználtsága általában már nem lesz annyira jó, mert ha egy folyamat adatai már nem férnének a lap végére, akkor azt nem lehet részekre bontani, hanem csak a következő lekérdezésben fog szerepelni. Külön jelezni kell, hogy az adott lapon már nincs több leíró,

offset		visszaadott tábla
0	→	{1, 2, 4, 5}
5	→	{7, 8, 11, 12}
12	→	{0, *, *, *}
11	→	{13, 15, 16, 19}
19	→	{20, 21, 22, 24}
24	→	{0, *, *, *}
22	→	{0, *, *, *}
21	→	{25, 26, 29, 30}
30	→	{-1, *, *, *}

6.2. táblázat. Az azonosítók listájának lekérdezése (tábla változása esetén)

viszont a processztábla még folytatódni fog; erre a 0-ás érték szolgál, ami így a lap végét jelzi.

A struktúrák összeállításának módja előtt sorra kell venni, hogy milyen mezőket is szeretnénk szerepeltetni. Mintaként a `ps` és `top` programok szolgálhatnak. A mezők és mezőnevek kiválasztásának szempontjai a következők voltak:

- A `ps` és a `top` program számára lehetőleg teljes funkcionalitást lehessen vele nyújtani.
- Minden mező csak egy néven szerepeljen (nem úgy mint a `ps` esetében); ne legyenek *alias*-ok.
- Ahol lehetséges inkább a kicsit hosszabb, de egyszerűbben értelmezhetőbb név szerepeljen, a hellyel elég lesz majd csak a kimenetnél takarékoskodni.
- Ne legyenek a nevek kis-nagybetű érzékenyek.
- Kizárólag az angol ábécé betűit használják, ne legyen bennük "_", "%" vagy "+" jel.

A mezőket nem lehet teljesen elkülönítve csoportokba sorolni, viszont érdemes a funkció szerint egymáshoz közelebb állókat együtt kezelni. A következőkben ezek részletes áttekintésére kerül sor.

Az első csoportba a folyamatok azonosítói tartoznak, az útvonalak, illetve az a parancs amivel a folyamat elindult (6.2 táblázat). A mezők típusát a kernelben használtnak megfelelően célszerű megadni, mivel a méretek architektúrától függően változhatnak, és elegendő csak újrafordítani a programot, nem kell a kódját módosítani. Néhány mező esetében az értékének a tárolása más típussal van megoldva a kernelben, mint a felhasználói programokban. Ez egyrészt az aktuális használati módnak, illetve az egységes kezelésnek köszönhető. Ezeknél meg kell győződni arról, hogy mekkora számtartomány ábrázolására van szükség; ha kérdéses lenne, akkor inkább a nagyobbat kell választani.

A második csoportba az ütemezés szempontjából lényeges jellemzők kerültek (6.2 táblázat), így az adatokat a processztáblából ki tudjuk olvasni. A maszkok esetében leginkább csak néhány bit értéke érdekes a lekérdezésekben, viszont az részekre bontása költségesebb lenne, mint ha egyben küldjük át a felhasználói programnak.

A további két csoport esetében is (6.2 és 6.2 táblázat) csak a mezők értelmezésében van különbség, az átvitel természetesen azonos módon oldható meg minden esetben. A lekérdezés leírásához elég mindössze a mezőnevekhez egy egyedi azonosítót rendelni. A számukból

mezőnév	leírás	típus
`command`	a kiadott parancs az összes argumentummal	`char[]`
`fsgid`	a fájlrendszer csoportjának azonosítója	`int`
`fsgroup`	a fájlrendszer csoportjának neve	`char[]`
`fsuid`	a fájlrendszer eléréséhez használt azonosító	`int`
`fsuser`	a fájlrendszer eléréséhez használt név	`char[]`
`gid`	csoport azonosítója	`int`
`group`	csoport neve	`char[]`
`pid`	folyamat azonosítója	`pid_t`
`ppid`	a szülőfolyamat azonosítója	`pid_t`
`root`	a folyamatból látható gyökérjegyzék	`char[]`
`ruid`	valódi felhasználó azonosító	`int`
`ruser`	valódi felhasználónév	`char[]`
`session`	a *session* azonosítója	`int`
`sgid`	az elmentett csoportazonosító	`int`
`sgroup`	az elmentett csoportnév	`char[]`
`tty`	a folyamathoz tartozó terminál neve	`char[]`
`uid`	a folyamat tulajdonosának azonosítója	`int`
`user`	a folyamat tulajdonosának neve	`char[]`

6.3. táblázat. A folyamathoz tartozó azonosítók, parancsok és útvonalak.

mezőnév	leírás	típus
`blocked`	a blokkolt szignálok maszkja	`long`
`caught`	az elkapott jelzések maszkja	`long`
`class`	az ütemezési osztály, amibe a folyamat tartozik	`int`
`flags`	a folyamat állapotához tartozó jelzőbitek	`long`
`ignored`	a figyelmen kívül hagyott jelzések maszkja	`long`
`nice`	az ütemezéshez használt *nice* érték	`int`
`pending`	a függőben lévő jelzések maszkja	`long`
`priority`	a folyamat prioritása	`int`
`sched`	ütemezési mód	`int`
`state`	a folyamat állapota	`int`
`wchan`	a függvény neve, amelyben várakozik	`char[]`

6.4. táblázat. Jelzések kezelése, állapotjellemzők és ütemezés.

mezőnév	leírás	típus
cpu	a processzorhasználat százalékos formában	int
elapsed	a folyamat indítása óta eltelt idő	long
lastcpu	az utoljára használt processzor	int
nthread	a folyamathoz tartozó szálak száma	int
pgid	a folyamat csoport azonosítója	int
psr	a processzor, amihez a folyamat éppen tartozik	int
start	a folyamat indításának ideje	long
thread	a szálak azonosítója	int
time	összesített processzoridő	long

6.5. táblázat. Futási idővel, processzorhasználattal, és szálkezeléssel kapcsolatos mezők.

mezőnév	leírás	típus
code	a folyamathoz tartozó kód mérete	int
data	az adat és a verem mérete	int
eip	utasítás mutatója	long
esp	verem mutatója	long
mem	a rezidens memóriahasználat	int
pagefault	a laphibák száma	int
res	a folyamat mérete a memóriában	int
shared	megosztott memória mérete	int
swap	a folyamatból kilapozott részének mérete	int
virt	a virtuális memória mérete	int

6.6. táblázat. Memória és lapozófájlhasználat.

látható, hogy erre egy bájt elegendő. A lekérdezett adatok utána már könnyen kezelhetőek, mivel vagy rögzített hosszúságú számértéknek (az adott architektúrának megfelelően persze), vagy pedig a C programozási nyelvben hagyományosan használt 0 értékkel záródó karakteres típus.

6.3. *További optimalizálási lehetőségek*

Feltételezhetjük, hogy a lekérdezések a teljes processztáblára vonatkoznak. Ekkor, ha a lekérdezés még nem ért el az utolsó folyamathoz, de a lap már betelt, akkor várható, hogy következni fog majd egy lekérdezés, amiben már tudjuk, hogy milyen `PID` fog szerepelni, illetve a hozzá tartozó `task_struct` mutató is könnyen hozzáférhető, mivel éppen ott tart a lekérdezés. A következő lekérdezésben a modulnak a kernel hasítótáblájában kell megkeresnie a mutatót, ami nagy mennyiségű folyamat esetén lassabb, mint ha azt előre külön letárolnánk. Az utoljára kiolvasott `PID`-et és a leíró struktúra mutatóját kell tehát együtt tárolni egy rövidebb listában, és a következő olvasásnál először ebben megnézni, hogy szerepel-e benne

az azonosító. A méretét úgy érdemes megválasztani, hogy annyi bejegyzésnek legyen benne hely, mint amennyi monitorozó programot szeretnénk használni.

A konkurrens hozzáférés nem akadályozza ennek a működését, mivel ha két (vagy akár több) folyamat éppen ugyanott tart a folyamatok olvasásában, akkor is csak a bejegyzések száma nő. Amire még ügyelni kell, hogy ha a várt lekérdezés mégsem történne meg, akkor a bejegyzés bennemarad a listában, és az így betelhet.

6.4. *Módosított felhasználói programok*

A lekérdezés már megoldottnak tekinthető, viszont el kell készíteni azokat a felhasználói programokat is, amikkel kényelmesen, parancssorból meg tudjuk figyelni a folyamatokat. Ezeknek a proc fájlrendszerben még bizonyos dolgokat elő kell készíteni, hogy a programoknak valóban csak a felhasználói bemenet kezelésével, az esetleges szűrésekkel és a megjelenítéssel kelljen foglalkozniuk, az adatok szabályozott átvitelével már ne.

Az út átviteli mód használatbavételének alapvetően két módját választhatjuk:

- Átírhatjuk a már létező programokat, hogy felhasználói szemszögből csak a sebességben legyen változás. Ez vélhetően kevesebb kód írásával jár, viszont alkalmazkodni kell a meglévő programok szerkezetéhez.
- Új programok írásába kezdünk, ami így teljes mértékben ki tudja majd használni az interfész adottságait.

Mindkét megoldásnak megvannak tehát az előnyei és hátrányai. Kompromisszumos megoldásként az új programban felhasználva a már létező programok bizonyos részeit könnyebben kezelhető kódot kapunk, és az alapvető műveletek ismételt megvalósításával sem kell foglalkoznunk.

A programok számára még egy további réteget is beiktathatunk. Ez lesz majd azért felelős, hogy több monitorozó program is tudja használni a modult párhuzamosan, és hogy a lekérdezés megadása is ellenőrzött módon történjen.

6.5. *A `/proc/query` jegyzék*

A lekérdezés módját (tehát lényegében azt, hogy milyen mezők szerepeljenek majd az eredménylistában) valahogy közölni kell az adatokat szolgáltató modullal. Az egyes lapok lekérdezéséhez elegendő volt egyetlen számértéket (az utolsó kiolvasott `PID` értéket) átadni, de több adat átvitelére már nem alkalmas az az interfész. A mezők kiválogatását mindenképpen a kerneloldalon kell megvalósítani, mivel ez jelenti az egyik legnagyobb teljesítménybeli többletet a hagyományos megoldással szemben.

A kézenfekvő megoldást a proc fájlrendszernek a fájlokba írási lehetősége adja. A fájlokat írni, és olvasni is lehet, ha a `file_operations` struktúrában beállítjuk a megfelelő függvényeket. A fájlba beírva a lekérdezést, az már tudni fogja, hogy a következő olvasásnál milyen kimenetet szolgáltasson. Ez ebben a formában azért lenne rossz megoldás, mert feltételezzük, hogy egyetlen monitorozó programról van szó, viszont többre is fel kell készülni. A gyakorlatban könnyen előfordulhat, olyan szituáció, hogy

- az első program beírja a lekérdezését,
- a második program beírja a lekérdezését,
- az első program kiolvassa az adatokat,

- a második program kiolvassa az adatokat.

A harmadik lépésnél a program nem feltétlenül olyan adatokat fog kapni, mint amiket várna. Az adatok átadásához használt protokollnál a rendelkezésre álló lehetőségek miatt szükséges volt, hogy a modul csak az offset alapján adja vissza az eredményeket, ne igényeljen egyéb információkat a lekérdezést végző programtól. Az olvasásnál a fájlt megnyitó folyamatról alig tud valamit a modul, és az eddigiekhez nem is volt többre szüksége. Egy bejegyzés tehát nem tünik elegendőnek ahhoz, hogy ki tudjon szolgálni több monitorozó folyamatot is.

A monitorozásnál a gyakori lekérdezések jelentik a nagyobb problémát. Ha csak az aktuális folyamatlistára vagyunk kíváncsiak, akkor azt, még ha kicsit késve is de megkapjuk, és utána megkereshetjük benne a számunkra lényeges részeket, viszont ha adott időközönként végezzük a lekérdezést, és nagyobb terhelés mellett a lekérdezéshez szükséges idő hosszabb, mint amilyen időközönként azt el szeretnénk végezni, az már komolyabb gondot okoz. Az utóbbi esetben elég lenne csak a monitorozás kezdetekor beállítani a lekérdezést, utána pedig az adott struktúrában várni az adatokat. El kell tehát különíteni egy inicializációs fázist, amelyben a program beállítja a lekérdezést, illetve biztosítani kell számára azt, hogy külön bejegyzést kapjon, amiből ki tudja olvasni az adatokat.

A megoldást egy, a monitorozó programok számára fenntartott jegyzék jelenti a `/proc` jegyzéken belül. Ez a `query` nevet kapta, és benne kétféle fájl található:

- a lekérdezés megadásához használható `create` nevű fájl,
- a monitorozó programok `PID`-jével megegyező nevű fájlok.

A `create` fájlt megnyitva létrejön a `query` jegyzékben a program `PID`-jével megegyező nevű fájl, amibe azután már beírhatja a lekérdezését, illetve lekérdezheti belőle az adatokat. A `PID` használatának itt az az előnye, hogy a bejegyzések így biztosan egyediek lesznek, illetve a felesleges bejegyzések is egyszerűen kiszűrhetők azzal, ha végignézzük a processztáblát, és kitöröljük azokat, amelyekhez már nem találjuk meg a folyamatot.

A folyamat azonosítóját már rögtön a fájl megnyitásakor egyszerűen megkaphatjuk, mivel csak az éppen futó folyamat azonosítóját kell felhhasználni, így a bejegyzésnek a nevét a `current->pid` értéke adja.

Azt illetően, hogy a folyamatok mikor fogják a `create` fájlt bezárni, nem sok feltételezéssel élhetünk. A használatához elegendő megnyitni és rögtön utána be is zárni azt, viszont fel kell készülni arra is, hogy a megnyitások átlapolódhatnak, így a bejegyzés ismételt létrehozása már nem szükséges. Erre azért is kell külön figyelmet fordítani, mivel a proc fájlrendszer esetében a fájlok kezelése nem annyira szigorú (mivel eleve a kernelben kezeli őket), és ezért könnyen előfordulhat, hogy egy jegyzékben több azonos nevű fájl is létrejön.

Minden új bejegyzésnek be kell állítani az íráshoz és olvasáshoz használandó függvényeit. Azért, hogy ezeket ne kelljen egyedileg létrehozni a lekérdezéshez tartozó információkat tárolni kell bennük. A `read_proc()` függvény `data` paramétere pont erre szolgált, viszont `file_operations`-ban lévő `read()` függvénynek nincs ilyen. Ebben a `file` struktúra `private_data` mezője lesz alkalmas arra, hogy a benne tárolt információ alapján meg lehessen különböztetni a fájlokat. Ebbe a lekérdezéshez tartozó leíró is megadható.

Ideális esetben a monitorozó programnak leállításakor a bejegyzést el kell távolítania. Ez tulajdonképpen azt jelenti, hogy az új lekérdezésnek az üres kimenetet adja meg, amiből így a modul tudja, hogy arra már nincs szükség.

7. fejezet

LEKÉRDEZÉSI IDŐK ÖSSZEHASONLÍTÁSA

A fejezet célja, hogy összehasonítsa a jelenleg használatban lévő eszközök, és az új lekérdezési módszer hatékonyságát. A tesztek különböző típusú és mértékű rendszerterhelés mellett készültek, szimulálandó az éles környezetben is előforduló problémás helyzeteket.

A tesztelési környezetet egy hagyományos PC jelentette, melynek fő paraméterei:

- Intel Celeron 900 típusú 2.2 GHz-es processzor
- 2 GB RAM
- Arch Linux operációs rendszer

A tesztesetekhez egy-egy shell-szkript tartozik. Az eredmények ilyen formában könnyedén reprodukálhatóak, módosíthatóak, és vihetők át más rendszerekre is. A futási időt lemérni a `time` parancs segítségével lehet. Ezzel meg lehet vizsgálni, hogy a program mennyi ideig futott (`real`), illetve hogy összesen mennyi processzoridőt használt el kernel- (`sys`) és felhasználói (`user`) módban.

A tesztekben a viszonyítási pontot a `ps` program adja. Ennek a paraméterezése az alábbi:

```
ps -eo pid,uid,s,cputime,comm
```

Az adatok összeállítását a kerneloldalon mindenképpen el kell végezni. Ez a proc fájlrendszerben lévő bejegyzés számára, illetve a gyorsabb lekérdezést segítő modul számára is ugyanannyi időbe telik. Ami az előnyt jelenti az új megoldásban, hogy a feldolgozási és átviteli fázis leegyszerűsödik, mivel nem kell az adatokat oda-vissza alakítani, és ahol megoldható ott bináris formában jelennek meg. A kernelen belül a számítások számán így tehát jelentősen nem lehet csökkenteni, tehát a tesztelés szempontjából tetszőleges lekérdezés megfelelő lenne. A választás azért éppen erre a lekérdezési módra esett, mert a gyakorlatban az egyszerűbb lekérdezések közül ez, vagy ehhez nagyon hasonló lekérdezések fordulnak elő.

7.1. *Nagy számú processz kezelése*

Az első tesztben azt lehet megvizsgálni, hogy hogyan befolyásolja a lekérdezés sebességét a lefoglalt azonosítók száma. A folyamatokat az alábbi szkript hozza létre:

```
for i in `seq 1 $1`
do
  sleep 10m &
done
```

Ebben a `$1` az első paramétere a szkriptnek, amivel meg lehet adni azt, hogy mennyi új folyamat jöjjön létre. Ezek mindegyike csak várakozni fog a megadott ideig (jelen esetben 10 percig). Ez után már meg lehet nézni, hogy a monitorozást végző programok milyen gyorsan tudják listázni ezeket.

A lekérdezés eredményének a kiírása is számításigényes, viszont terminálonként változó, hogy mennyire. Ez befolyásolhatja az eredményeket. Ezt elkerülendő a programok kimenete minden tesztnél a `/dev/null`-ba van átirányítva.

Azért, hogy minél pontosabb képet kapjunk arról, hogy a processzek számának növekedésével hogyan nő a monitorozó programok válaszideje több mérést is el kell végezni. A tesztek során egy beállításhoz 50 tesztlekérdezés készült, és a becsült értékeket a mért idők átlaga adja.

A mérések eredményei külön táblázatba kerültek (7.1 táblázat). Ennek első oszlopában az szerepel, hogy a mérésnél mennyi folyamat volt a processztáblában, a második oszlopban a `ps` futásideje látható, az utolsó négy oszlopban pedig az új lekérdezési módszer eredményei a lapok mérete szerint. A futási idők másodpercben vannak megadva.

processzek száma	`ps`	8 kb	16 kb	32 kb	64 kb
2500	0.135	0.006	0.005	0.006	0.005
5000	0.262	0.009	0.009	0.009	0.009
7500	0.392	0.013	0.012	0.013	0.012
10000	0.527	0.016	0.016	0.017	0.016
12500	0.673	0.02	0.019	0.02	0.019
15000	0.797	0.03	0.026	0.027	0.03

7.1. táblázat. Az átlagos lekérdezési idők nagy számú folyamat esetén

Ebből nyilvánvalóan látszik, hogy a blokkos átvitel jelentősen gyorsabb, mint ha az adatokat a `/proc` jegyzékből olvasnánk ki. Az is látszik, hogy a lapok mérete jelentősen nem befolyásolja a lekérdezés sebességét, van néhány eset amikor még lassabb eredményt is produkál. Ebben közrejátszhatnak a háttérben futó rendszerfolyamatok, a kernelben a memórialapok másolási módja, vagy esetleges mérési pontatlanságok is.

7.2. *Intenzív CPU terhelés*

Nagy processzorterhelés többek között matematikai számítások jelenthetnek, mint például amit az

```
echo "88888888 ^ 88888888 | bc"
```

parancs kiadásával indíthatunk el.

A teszteket az előző mintájára végezhetjük, viszont itt már nem szükséges annyira sok folyamatot létrehozni, elegendő 1000 számításigényes processz is, hogy értékelhető eredményt szolgáltasson a lekérdezések futási idejéről.

A programok kimenete szintén a `/dev/null` fájlba volt átirányítva, így csak azt mértük, hogy a lekérdezés összeállítása mennyi időt vett igénybe, a kimenet megjelenítésével már nem. A futtatások eredménye a 7.2 táblázatban szerepel.

Ezek az adatok szintén azt mutatják, hogy ilyen esetben is sokkal kevesebb ideig tart a lekérdezés.

processzek száma	ps	8 kb	16 kb	32 kb	64 kb
1000	42.3	2.4	1.5	1.2	0.8

7.2. táblázat. Az átlagos lekérdezési idők nagy számú folyamat esetén

7.3. *Intenzív I/O terhelés*

Folyamatosan nagy I/O terhelést például úgy kaphatunk, ha különböző partíciókon lévő adatokat mozgatunk oda-vissza.

```
for i in `seq 1 $1`
do
        mv /part1/Huge.dat /part2/Huge.dat
        mv /part2/Huge.dat /part1/Huge.dat
done
```

Ennek a vizsgálata azért lényeges, mert a proc fájlrendszer is érzékeny lehet az I/O műveletekre. A viszonyítási alapot itt az jelentheti, ha megnézzük, hogy a lekérdezések milyen futási időket adnak a terhelés nélkül. Normális terhelés mellett a `ps` futási ideje átlagosan 0.013 másodperc körül van, az új megoldásban pedig ez 0.001. A terhelés mellett kapott futási időket 7.3 táblázat foglalja össze.

ps	8 kb	16 kb	32 kb	64 kb
0.23	0.004	0.001	0.001	0.001

7.3. táblázat. Az átlagos lekérdezési idők I/O terhelés mellett.

Az új interfész ebben az esetben is gyorsabb hozzáférést tett lehetővé. A lapméret növelése néhány teszt esetében okozott némi javulást, viszont csak elhanyagolható mértékben.

7.4. *Összegzés*

Mint a mérésekből kiderült, ezzel a lekérdezési módszerrel gyorsítani lehet a felhasználói programokat. Az interfész implementálához a kernel különféle szolgáltatásait kell használni egyidejűleg. Ez problémát jelenthet, mivel a kernel folyamatosan változik, és emiatt a programokat aktualizálni kell az újabb verziókhoz.

Az új megoldás a lekérdezések sebességének javításán túl arra is törekszik, hogy egy egységesebb módot adjon a folyamatok jellemzőinek kezeléséhez, mint az a ps vagy `top` programok esetében jelenleg van. Ennek előnyei az interfészhez készített felhasználói programok írásakor, illetve azok használatakor is megmutatkozik.

Az átviteli módot még lehet optimalizálni, illetve készíthetők további tesztek arra vonatkozóan, hogy ténylegesen mennyivel is lesz hatékonyabb ez a megoldás, viszont az világosan látszik, hogy a dolgozatban bemutatott megoldási módokkal ugyanaz az eredmény kevesebb erőforrással is előállítható.

A *www.iit.uni-miskolc.hu/~piller/proc_query* honlapon elérhetőek a dolgozatban bemutatott programok illetve tesztek forráskódjai.

Irodalomjegyzék

[1] Jaroslav Šoltýs: *Linux Kernel 2.6 Documentation*, Master thesis, Bratislava, 2006.

[2] Greg Kroah-Hartman: *Linux Kernel in a Nutshell*, O'Reilly, 2006.

[3] *The Linux Documentation Project*, `tldp.org`

[4] Erik (J. A. K.) Mouw: *Linux Kernel Procfs Guide*,
Delft University of Technology Faculty of Information Technology and Systems, 2001.

[5] Avinesh Kumar: *TASK_KILLABLE: New process state in Linux*,
IBM Software Labs in Pune, India, 2008.

[6] *Linux Cross Reference*, `http://lxr.free-electrons.com/`

[7] Internet Engineering Task Force: *Request For Comments (RFC)*,
`http://www.ietf.org/`

[8] Alessandro Rubini, *Kernel Korner: The sysctl Interface*,
Linux Journal, volume 41., 1997.

[9] Moshe Bar, *Kernel Korner: The Linux Scheduler*,
Linux Journal, volume 72., 2000.

[10] Abhishek Nayani, Mel Gorman, Rodrigo S. de Castro, *Memory Management in Linux*

[11] M. Tim Jones, *Anatomy of the Linux file system*
`http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/`

[12] Moshe Bar, *The Linux Signals Handling Model*,
Linux Journal, volume 73., 2000.