Operációs rendszerek felépítése

Gyakorlati útmutató
Készítette: Horányi Gergő, Micskei Zoltán
Utolsó módosítás: 2014.02.19.

A labor célja, hogy megismerkedjünk az operációs rendszerek felépítésével kapcsolatos általános fogalmakkal (kernel, védett és felhasználói mód, rendszerhívás, stb.), és néhány egyszerű eszköz segítségével megvizsgáljuk ezek megvalósítását Linux és Windows esetén.

1 Linux útmutató

A feladatok megoldásához egy VMware virtuális gépbe telepített Linux operációs rendszert fogunk használni. A laborfeladatokat Ubuntu disztribúción próbáltuk ki, de elvileg más disztribúción is könnyedén elvégezhetőek a feladatok.

1.1 Linux rendszerhívások

Az első feladatban azt vizsgáljuk meg, hogy egy egyszerű program milyen rendszerhívásokat használ, és azokat hogyan éri el.

1. Indítsuk el a virtuális gépet!

   Lépjünk be, majd töltsük le a tárgy weboldaláról a laborhoz tartozó kódrészleteket, majd csomagoljuk ki azokat.

2. Indítsuk el a Terminal alkalmazást!

   Lépjünk be a *Feladat1* könyvtárba:

   ```
   cd <kodreszletek eleresi utja>/linux/Feladat1
   ```

   A könyvtár tartalma egy *main.c* fájl, amely egy nagyon egyszerű program:

   ```
   #include <stdio.h>

   int main() {
       fopen("main.c", "r");
       return 0;
   }
   ```

3. Fordítsuk le a programot!

   ```
   gcc -g -o program main.c
   ```

   A `-g` paraméter azért szükséges, hogy a fordított kódba debug információk is kerüljenek. A program innentől futtatható a `./program` utasítással.

4. Gondoljuk végig, hogy fog-e a lefordított program rendszerhívásokat használni?
    - Ha igen, akkor miért?
5. Vizsgáljuk meg, hogy milyen rendszerhívásokat használ ténylegesen a program!

    Ehhez a Linux *strace* utasítása használható. Ehelyett azonban használjuk most az *ltrace* utasítást, mert így együtt láthatjuk a könyvtárhívásokat és a rendszerhívásokat is:

    ```
    ltrace -S ./program
    ```

    Erre számítottunk?
    - Mi okozza ezt a rengeteg rendszerhívást?
    - Mi lehet az *mmap2()* hívás?
    - Hol indul maga a main függvény?

    A rendszerhívások értelmezésében segíthet a `man` parancs.
6. Az *fopen()* hívás jól látható, de mintha megszakadna a futása.
    - Mit gondolunk, miért „szakadt meg” a futása?
7. Vizsgáljuk meg egy debugger (gdb) segítségével is a programot!

    Töltsük be a gdb-be a programot!

    ```
    gdb ./program
    ```

    Ugorjunk a programban oda, ahol a main függvény ténylegesen elindul!

    ```
    break main
    run
    ```

    Állítsuk be, hogy a gdb elkapja az *open* rendszerhívást, majd futtassuk tovább a programot!

    ```
    catch syscall open
    continue
    ```

    Vizsgáljuk meg, hogy éppen milyen függvények vannak meghívva!

    ```
    backtrace
    ```

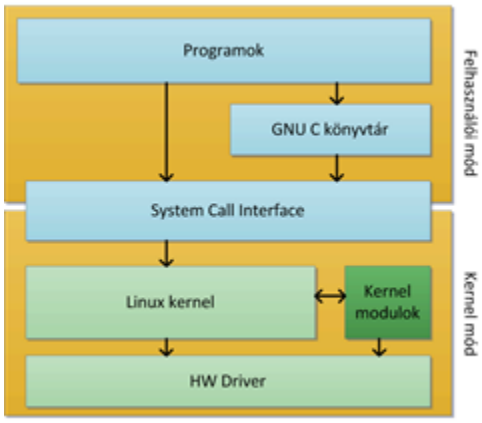
    - Mi lehet a libc.so.6 file?
    - Mi lehet a __kernel_vsyscall() függvény és miért nem látunk „mélyebbre“?

1.2 Kernelmodul készítése

Az előző feladatban megvizsgáltuk, hogy egy felhasználói módban futó alkalmazás hogyan kommunikál a kernellel. Ebben a feladatban kernel módban futó alkalmazást fogunk létrehozni: kernelmodult fogunk készíteni.

A kernelmodulok a kernelt kibővítik (*Mac OS X alatt Kernel Extension a nevük*), és a kernel részeként futhatnak. Olyan utasításokat adhatnak ki, amelyeket egy felhasználói program nem tehet meg és olyan adatstruktúrákhoz is hozzáférnek, amelyek a felhasználói módból rejtettek (és csak rendszerhívásokon keresztül vagy még úgy sem érhetőek el).

A Linux nagyon leegyszerűsített architektúrája az 1. ábrán látható.



1. ábra: A Linux egyszerűsített felépítése

1. Lépjünk át a Feladat2 könyvtárba!

```
cd <kodreszletek eleresi utja>/linux/Feladat2
```

2. Nézzük meg a *mymodule.c* állományt!

```
cat mymodule.c
```

A mymodule.c állomány tartalma:

```
#include <linux/module.h>
#include <linux/kernel.h>

int start_mymodule(void) {
    printk(KERN_ALERT "MyModule kernelmodul betoltodott!\n");
    return 0;
}

void exit_mymodule(void) {
    printk(KERN_ALERT "MyModule kernelmodul eltavolitva.\n");
}

module_init(start_mymodule);
```

```
module_exit(exit_mymodule);
```

Láthatjuk, hogy a program hivatkozik néhány linux könyvtárban található fejlécre. Ezek szükségesek a kernelmodulokhoz.

A *printk* utasítás a *printf* kernelbeli változata, hiszen a kernel nem használhatja a C függvénykönyvtárat. A *KERN_ALERT* makró az üzenet prioritását jelzi. Az üzenetek később a *dmesg* utasítással olvashatóak.

3. Fordítsuk le a modult!

   Ebben segítségünkre lesz egy makefile, amelynek a tartalmára érdemes egy pillantást vetni.

   ```
   cat Makefile
   ```

   A fordítást ezekkel az utasításokkal végzi:

   ```
   obj-m += mymodule.o

   all:
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
   clean:
    $(MAKE) -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
   ```

   A fordításhoz indítsuk el az alábbi parancsot:

   ```
   make
   ```

4. A lefordított modult töltsük be és nézzük meg a működését!

   ```
   sudo insmod mymodule.ko
   ```

   A betöltést a `dmesg` paranccsal ellenőrizhetjük.

   ```
   dmesg
   ```

   Az eltávolításhoz az alábbi parancsot adjuk ki:

   ```
   sudo rmmod mymodule
   ```

5. Mi történik, ha a kernelmodulba egy végtelen ciklus kerül? Próbáljuk ki!

   Írjuk át a mymodule.c állományt.

   ```
   nano mymodule.c
   ```

   A start_mymodule függvény visszatérése elé írjuk be az alábbi utasítást:

   ```
   while(1);
   ```

   Mentsük el, fordítsuk le újra és töltsük be!

```
make
sudo insmod mymodule.ko
```

- Mi lesz az eredménye a kernelmodulban okozott végtelen ciklusnak?

6. Csináljunk kernel panicot!

   Ehhez a teendőnk csak annyi, hogy a végtelen ciklus helyére írjuk az alábbi utasítást:

   ```
   panic("Panik!!!");
   ```

   - Ezután persze fordítsuk le újra és töltsük be!
   - Mi történik most? Van különbség?

2 Windows útmutató

A feladatokat egy Windows 8.1 virtuális gépen érdemes végrehajtani, amire fel kell telepíteni a szükséges alkalmazások (Visual Studio 2013, Windows SDK 8.1, Windows Driver Kit 8.1, Sysinternals Suite legfrissebb verziója). Ehhez segítséget a Windows alapok anyaga ad.

2.1 Windows rendszerhívások

A következő feladatban a Windows API rendszerhívásainak használatát fogjuk megvizsgálni.

1. Indítsuk el a Windows virtuális gépet. Töltsük le és tömörítsük ki a laborhoz tartozó kódrészleteket a `c:\code` könyvtárba.
2. Nézzük meg a `C:\code\windows\windows-api` könyvtárban lévő *Simple.c* forrásfájlt.

   A program a CreateFile, ReadFile és CloseHandle API hívásokat használja:

   ```
   hFile = CreateFile("ReadMe.txt", GENERIC_READ|GENERIC_WRITE,
    FILE_SHARE_READ, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

   ReadFile(hFile,strVal,512,&wmWritten,NULL);
   ...
   CloseHandle(hFile);
   ```

3. Fordítsuk le a programot!

   Nyissunk egy *Developer Command Prompt*ot (ebben többek között a PATH-ban be vannak állítva az SDK és VS könyvtárai), majd fordítsuk le:

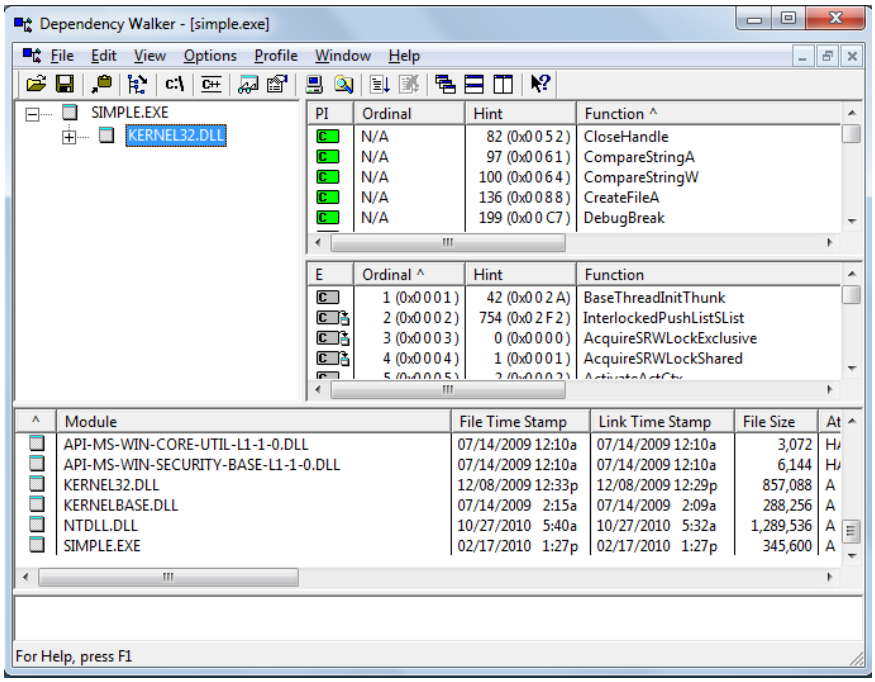
   ```
   cl /Zi Simple.c
   ```

   Ennek hatására a C fordító (cl.exe) létrehozza a forrásfájlból az object fájlt, majd a linker (link.exe) elkészíti a végrehajtható állományt.

4. Nézzük meg a program függőségeit!

   A *Dependency Walker*[1] segítségével vizsgáljuk meg, hogy milyen könyvtárakra, és azon belül milyen függvényekre hivatkozik a Simple.exe nevű programunk.

[1] Dependency Walker, http://www.dependencywalker.com/



2. ábra: Dependency Walker

A bal felső ablakban egy faszerkezetben látjuk a programunkból hivatkozott DLL-ek listáját, valamint továbbkövethetjük, hogy azok mire hivatkoznak. A jobb felső részben (PI – Parent Import) azok a függvények szerepelnek, amiket az aktuálisan kijelölt könyvtárból a rá hivatkozók használnak. A jobb középső részen (E – Export) az aktuálisan kijelölt könyvtár által exportált összes függvény látszik. Végül az alsó részen az adott környezetben hivatkozott összes modul listája látható, a modulok részletes adataival.

- Nézzük meg, hogy a simple.exe milyen API hívásokat használ a kernel32.dll-ből (a windowsos alrendszerhez tartozó DLL)! Hány másik függvényt látunk a simple.c forrásfájlban szereplő 3 rendszerhíváson kívül?
- Nézzük meg, hogy milyen függőségei vannak a kernel32.dll-nek! Mit ismerünk fel ezek közül?
- Keressük ki az NTDLL.DLL-t! Mi ennek a szerepe? Görgessük végig az általa exportált függvényeket, hogy képet kapjunk az NT API-ról!

5. Kövessük végig egy rendszerhívás menetét debuggerben!
   - Indítsuk el a *WinDbg* programot!
   - Nyissuk meg benne a simple.exe fájlt (Ctrl + E)!
   - Nézzük meg, hogy hol tart jelenleg a végrehajtás (a k-val kezdődő parancsok az aktuális szálhoz tartozó verem tartalmát listázzák mindig ki):

```
k
```

```
0:000> k
ChildEBP RetAddr
002ef97c 775be376 ntdll!LdrpDoDebuggerBreak+0x2c
002efad8 775a8fc8 ntdll!LdrpInitializeProcess+0x124d
002efb28 7759b2f9 ntdll!_LdrpInitialize+0x78
002efb38 00000000 ntdll!LdrInitializeThunk+0x10
```

Helyezzünk el egy töréspontot a ZwReadFile függvényre, majd hagyjuk futni az alkalmazást:

```
bp ntdll!ZwReadFile
g
```

- Nézzük meg a verem állapotát, milyen függvényeken keresztül vezet a hívás?

A *tr* (trace) parancs segítségével kezdjük el léptetni a programunkat utasításonként.

- Mi történik a ZwReadFile-on belül?
- Miért csak ennyit látunk?

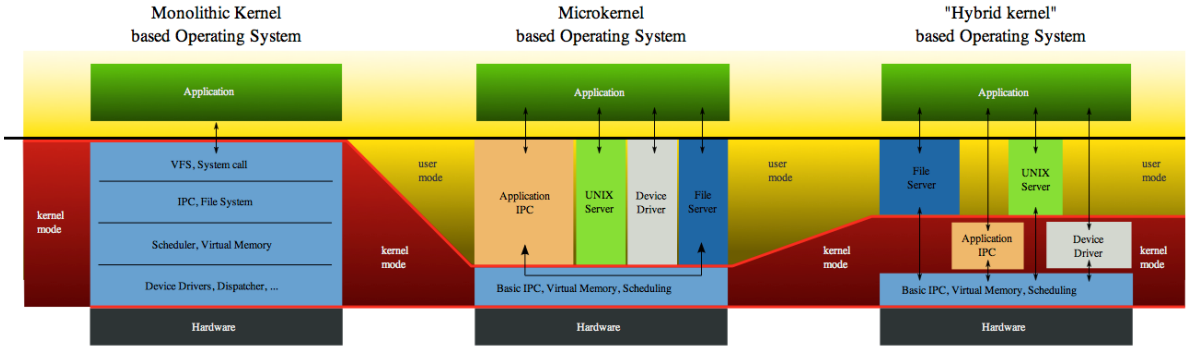
```
0:000> bp ntdll!NtReadFile
0:000> g
Breakpoint 0 hit
eax=0026f628 ebx=0026f8b4 ecx=00000000 edx=00000020 esi=00000000 edi=00000020
eip=77585560 esp=0026f5e8 ebp=0026f648 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!NtReadFile:
77585560 b811010000      mov     eax,111h
0:000> tr
ntdll!ZwReadFile+0x5:
77585565 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
0:000> t
ntdll!ZwReadFile+0xa:
7758556a ff12            call    dword ptr [edx]      ds:0023:7ffe0300={ntdll!KiFastSystemCall (77586340)}
0:000>
ntdll!KiFastSystemCall:
77586340 8bd4            mov     edx,esp
0:000> t
ntdll!KiFastSystemCall+0x2:
77586342 0f34            sysenter
0:000> t
ntdll!ZwReadFile+0xc:
7758556c c22400          ret     24h
0:000> t
KERNELBASE!ReadFile+0x118:
7576abad 3d03010000      cmp     eax,103h
```

(A WinDbg kimenetében az eax=… sorok a regiszterek aktuális tartalmát listázzák. A további sorok pedig azt jelzik, hogy éppen hol tartunk, pl. az ntdll!KiFastSystemCall+0x2 azt jelenti, hogy az ntdll modul KiFastSystemCall függvényének kezdőcíme plusz hexadecimális 2-n áll jelenleg az utasításszámláló. Az utána lévő sor a végrehajtandó utasítást adja meg <memóriacím> <opcode> <assembly utasítás> <paraméterek> formában.)

3 Mac OS X útmutató

A Mac OS X operációs rendszer azért kerül bemutatásra, mert a felépítése eltér a megszokottól. A Mac OS felépítésére általában hibrid kernelként szoktak hivatkozni.

Az alábbi ábra a három tipikus kernelarchitektúrát hasonlítja össze nagy vonalakban.



3. ábra: Kernelarchitektúrák (forrás: http://en.wikipedia.org/wiki/Hybrid_kernel)

A különbség jól látható: a különböző operációs rendszereknek különböző részei futnak kernel módban.

3.1 XNU: a Mac OS X magja

Az XNU kernel alapvető részei:

- Mach mikrokernel
- BSD réteg
- I/O Kit: a driverek futtatási környezete
- Libkern: a kernel egy belső könyvtára
- Libsa: kernel belső könyvtára a rendszerindításhoz
- The Platform Expert: hardver absztrakciós réteg (HAL)
- Kernel kiterjesztések

Mivel az XNU nyílt forráskódú, ezért bármikor megtekinthető a kernel forrása is: http://opensource.apple.com

3.1.1 Mach

Az XNU magja, amely a kritikus és alapvető szolgáltatásokat nyújtja a rendszer számára. A Mach kezeli a processzorokat, az ütemezést, a virtuális memóriát és az alacsonyszintű IPC (Inter-Process Communication) szolgáltatásokat.

3.1.2 BSD

A Mac OS X kernelben megtalálható egy FreeBSD-ből származtatott réteg. Ez nem azt jelenti, hogy az XNU-ban egy jól meghatározható részen (például egy Mach folyamatként futva) egy BSD kernel is fut. Bizonyos részek az eredetihez nagyon hasonló módon megtalálhatóak, de egyes részeket komolyan átdolgoztak a fejlesztők, hogy együttműködhessen a kernel többi részével.

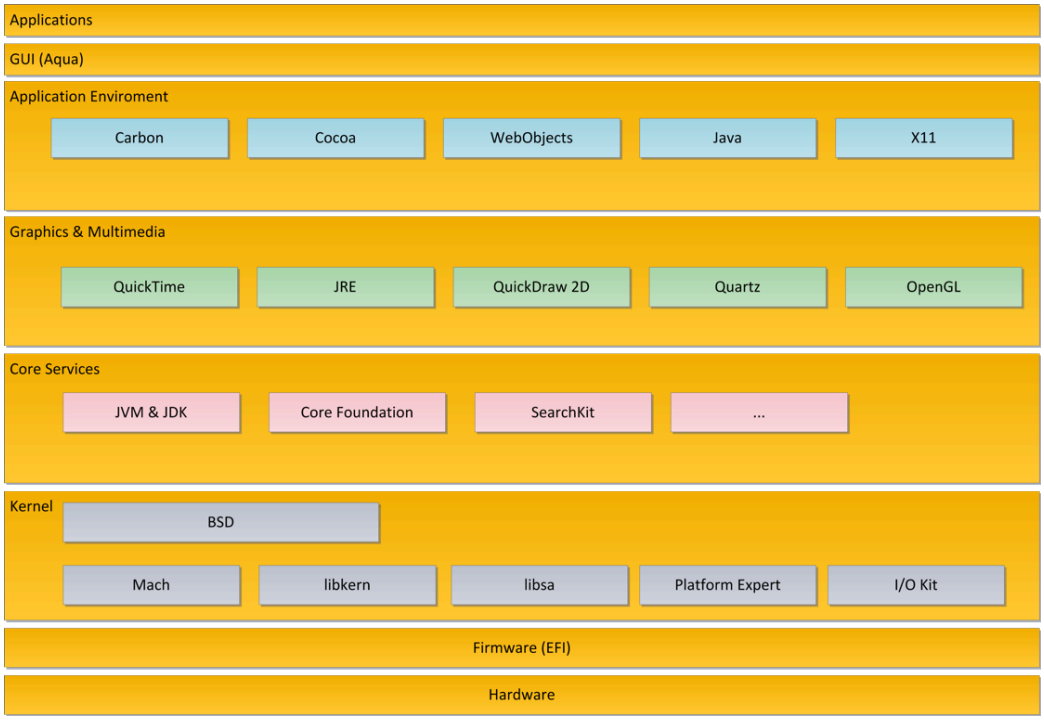
A BSD réteg kezeli a fájlrendszert, a hálózatot, a biztonságot. Ebben a rétegben van megvalósítva a POSIX API is.

3.1.3 I/O Kit

Az XNU-ban a driverek kezeléséért az I/O Kit felel. A feladata a Plug and Play támogatása, az on-demand driver betöltés, valamint a megfelelő energiagazdálkodási stratégiák kezelése is.

3.2 Mac OS X felépítése

Az alábbi ábra összefoglalja az előbb felsorolt komponensek viszonyát.



3.3 Rendszerhívások kezelése

Mivel az OS X-ben a kernel több különböző részre osztható, így érdemes lehet megfigyelni, hogy a rendszerhívásokat melyik alrendszer kezeli. A válasz: a BSD és a Mach együtt.

A rendszer *top* utasításával megfigyelhető, hogy egy-egy alkalmazás milyen rendszerhívásokból mennyit használt. Ezt mutatja az alábbi ábra.

```
Processes: 99 total, 3 running, 96 sleeping, 444 threads
Load Avg: 0.04, 0.07, 0.15  CPU usage: 3.70% user, 5.9% sys, 91.20% idle   SharedLibs: 5520K resident, 4640K data, 0B linkedit. MemRegions: 16856 total, 535M resident, 31M private,
PhysMem: 469M wired, 921M active, 813M inactive, 2202M used, 1893M free. VM: 208G vsize, 1040M framework vsize, 3189898(0) pageins, 175458(0) pageouts.  Networks: packets: 7421723/6
Disks: 646655/19G read, 700804/24G written.

PID    COMMAND          %CPU TIME     #TH  #WQ  #PORT #MREG RPRVT  RSHRD  RSIZE  VPRVT  VSIZE  PGRP  PPID  STATE    UID  FAULTS   COW    MSGSENT     MSGRECV     SYSBSD     SYSMACH
92619  screencapture    0.5  00:00.06 3    2    42    86    500K   23M    2896K  13M    2666M  235   1     sleeping 501  1572+    362+   1633+       610+        496        1538+
92611  top              6.7  00:00.74 1/1  0    25    36    2480K  276K   3076K  17M    2378M  92611 89029 running  0    9864+    56     400398+     200175+     3353+      201209+
92568  BetterTouchTool  0.3  00:00.34 4/1  2    100+  135   3316K  17M    11M    31M    2671M  92568 230   running  501  4729+    553    6729+       4526+       2894+      13268+
92507- Keynote          0.0  00:06.09 8    2    171   584   27M    103M   83M    53M    1130M  92507 230   sleeping 501  65130    2382   136186+     71331+      49191      123898+
92332- Google Chrome He 0.0  00:00.34 6    1    94    163   9512K  48M    24M    32M    987M   92320 92320 sleeping 501  7002     970    5448        2740        10325      5205
92331- Google Chrome He 0.0  00:00.29 5    1    92    152   7812K  48M    20M    38M    985M   92320 92320 sleeping 501  5936     959    4715        2353        9672       4427
92330- Google Chrome He 0.0  00:00.54 5    1    92    152   7160K  48M    20M    36M    983M   92320 92320 sleeping 501  6688     972    19703+      9992+       12815      16556+
92329- Google Chrome He 0.0  00:00.93 5    1    92    143   7200K+ 48M    19M+   44M    983M   92320 92320 sleeping 501  7602+    964    49075+      24864+      14201+     30229+
92328- Google Chrome He 0.0  00:00.26 5    1    93    142   7152K  48M    18M    44M    983M   92320 92320 sleeping 501  5374     976    4834+       2426+       7122       4389+
92326- Google Chrome He 0.0  00:00.39 5    1    97    252   13M    59M    27M    48M    1023M  92320 92320 sleeping 501  11314    993    3745        1818        7899       2527
92320- Google Chrome    0.0  00:03.81 22   1    198   331   36M    66M    63M    190M   1160M  92320 230   sleeping 501  32868    3087   34350+      15928+      108272+    87380+
92297- VMware Fusion He 0.0  00:00.95 4    1    167   150   6344K  15M    16M    36M    910M   92297 230   sleeping 501  5689     515    61904+      30710+      3421+      66564+
92263- Microsoft AU Dae 0.0  00:00.03 2    1    59    64    712K   640K   1856K  29M    854M   92263 230   sleeping 501  753      148    1370        685         714        983
92251- Microsoft Word   0.2  00:43.30 4    2    121   683   58M    74M    113M   165M   1649M  92251 230   sleeping 501  176651   1944   442750+     198790+     154072     436491+
92248  mdworker         0.0  00:01.04 3    1    50    105   5928K  14M    17M    42M    2424M  92248 1     sleeping 501  7075     165    2190        882         9229       1452
89370- Numbers          0.0  01:23.31 8    2    193   626   10M    107M   44M    56M    1126M  89370 230   sleeping 501  120052   2437   723064+     307515+     68978      722615+
89029  bash             0.0  00:00.13 1    0    17    25    288K   744K   884K   17M    2378M  89029 89028 sleeping 501  2057     1040   128         56          5985       95
89028  login            0.0  00:00.01 1    0    22    56    40K    324K   964K   19M    2379M  89028 85747 sleeping 0    667      118    288         133         1163       190
```

3.4 Dtrace

A *dtrace* a Sun Microsystems által 2005-ben kifejlesztett kiterjedt mérőrendszer. A Sun (ma már Oracle) a Solaris operációs rendszer számára fejlesztette ki, de ma már elérhető Mac OS X, FreeBSD és NetBSD alatt is. A dtrace segítségével nyomon követhetjük, hogy a rendszerben milyen függvényhívások történnek. Ebben a nagyon nagy számú mérőpont van segítségünkre. Egy mérőpont lehet egy függvény be- vagy kilépőpontja.

A `dtrace -l` hívással kilistáztathatjuk az összes ilyen pontot (ez azonban egy áttekinthetetlenül hosszú lista).

```
somimac:Release pumafi$ sudo dtrace -l | wc -l
  108347
```

A dtrace segítségével megvizsgálhatjuk, hogy egy program milyen rendszerhívásokat használ. Ehhez az alábbi scriptet kell megírni:

```
syscall:::entry
/pid == $1/ {}
syscall:::return
/pid == $1/ {}
```

Ha nyomon szeretnénk követni a fájl megnyitásához szükséges függvényeket, akkor a következő scriptet kell futtatni:

```
#pragma D option flowindent
syscall::open_nocancel:entry
/pid == $1/ {
 trace_me = 1;
 printf ("Arg: %s", copyinstr(arg0));
}
fbt:::entry
/trace_me == 1 && pid == $1/ {
}
fbt:::return
/trace_me == 1 && pid == $1/ {
}
syscall::open_nocancel:return
/pid == $1/ {
 trace_me = 0;
}
```

Mindkét script egy paramétert vár, amely a vizsgálandó alkalmazás azonosítója (PID).

A második script eredményének egy részlete:

```
CPU FUNCTION
  1  => open_nocancel                          Arg: program.c
  1    -> open_nocancel
  1      -> vfs_context_current
  1      <- vfs_context_current
  1      -> vfs_context_proc
  1        -> get_bsdthreadtask_info
  1        <- get_bsdthreadtask_info
  1      <- vfs_context_proc
  1      -> vfs_context_current
  1      <- vfs_context_current
  1      -> vfs_context_current
  1      <- vfs_context_current
  1      -> vfs_context_proc
  1        -> get_bsdthreadtask_info
  1        <- get_bsdthreadtask_info
  1      <- vfs_context_proc
  1      -> vfs_context_thread
  1      <- vfs_context_thread
  1      -> vfs_context_current
  1      <- vfs_context_current
  1      -> falloc
  1        -> lck_mtx_lock
  1        <- lck_mtx_lock
  1        -> falloc_locked
  1          -> fdalloc
  1          <- fdalloc
  1          -> proc_ucred
  1          <- proc_ucred
  1          -> mac_file_check_create
  1            -> mac_policy_list_conditional_busy
  1            <- mac_policy_list_conditional_busy
  1          <- mac_file_check_create
  1          -> lck_mtx_unlock_darwin10
  1          <- lck_mtx_unlock_darwin10
  1          -> _MALLOC_ZONE
  1          <- _MALLOC_ZONE
  1          -> zalloc
  1            -> zalloc_canblock
  1              -> lck_mtx_lock_spin
  1              <- lck_mtx_lock_spin
  1              -> lck_mtx_unlock_darwin10
  1              <- lck_mtx_unlock_darwin10
  1            <- zalloc_canblock
  1          <- zalloc
  1          -> _MALLOC_ZONE
  1          <- _MALLOC_ZONE
  1          -> zalloc
  1            -> zalloc_canblock
  1              -> lck_mtx_lock_spin
  1              <- lck_mtx_lock_spin
  1              -> lck_mtx_unlock_darwin10
  1              <- lck_mtx_unlock_darwin10
  1            <- zalloc_canblock
  1          <- zalloc
  1          -> lck_mtx_init
  1            -> lck_grp_reference
  1            <- lck_grp_reference
  1          <- lck_mtx_init
  1          -> lck_grp_lckcnt_incr
```

Ez rendkívül jól szemlélteti, hogy egy egyszerű fájlmegnyitás milyen bonyolult művelet a rendszer belsejében.

4 Összefoglalás

A labor során megnéztük, hogy általános célú operációs rendszerek esetén (Linux és Windows) hogyan lehet az operációs rendszer funkcióit egy felhasználói módú alkalmazásból elérni, valamint, hogy hogyan lehet egy egyszerű kernel modult készíteni és betölteni.

Kipróbáltunk pár hasznos eszközt (strace és ltrace, gdb, Dependency Walker, WinDbg), ezekre érdemes emlékezni.

Mindkét platform esetén természetesen csak a legalapvetőbb funkciókat próbáltuk ki, az útmutató végén szereplő linkeken lehet további információt találni a témában.

5 Források

Linux

[1] Bryan Henderson: Linux Loadable Kernel Module HOWTO
http://tldp.org/HOWTO/Module-HOWTO/

[2] GDB: The GNU Project Debugger
http://www.gnu.org/software/gdb/

[3] William B. Zimmerly: Fun with strace and the GDB Debugger
http://www.ibm.com/developerworks/aix/library/au-unix-strace.html

Windows

[4] Mike Taulty's Blog : A word for WinDbg,
http://mtaulty.com/communityserver/blogs/mike_taultys_blog/archive/2004/08/03/4656.aspx

[5] Frequently used Debugger commands, http://www.tonyschr.net/debugging.htm

[6] Don Burn. Getting Started with the Windows Driver Development Environment, 2010,
http://www.microsoft.com/whdc/driver/foundation/drvdev_intro.mspx

Mac OS

[7] Apple: Kernel Programming Guide,
http://developer.apple.com/library/mac/#documentation/Darwin/Conceptual/KernelProgramming/About/About.html

[8] Amit Singh: Mac OS X Internals, A Systems Approach, Addison-Wesley, 2007

[9] Greg Miller: Exploring Leopard with Dtrace,
http://www.mactech.com/articles/mactech/Vol.23/23.11/ExploringLeopardwithDTrace/index.html

[10] Apple: Instruments User Guide,
http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/Introduction/Introduction.html

[11] Mac Dev Center, http://developer.apple.com/devcenter/mac/index.action

Magyarázatok

Segítségképp itt közlünk néhány magyarázatot a labor kérdéseihez.

- 1.1, 6. pont, fopen futásának megszakadása: Amikor egy program rendszerhívást akar kezdeményezni, akkor meg kell hívnia egy wrapper függvényt (ezt teszi meg a C könyvtár valahol az fopen belsejében). A függvény hatására a kernel „elaltatja“ az

aktuális programot (SYS_brk), végrehajtja a rendszerhívást, majd visszaadja a futást a programnak. A program az egészről nem is értesül (hiszen amíg a rendszerhívás végrehajtódik, a programunk nem fut), így a rendszerhívások teljesen transzparensen hajtódnak végre. Látható azonban, hogy ez jóval bonyolultabb (és lassabb) feladat, mint egy egyszerű függvényhívás.

- 1.1, 7. pont, kernel_vsyscall(): A függvény a rendszerhívások fogadásáért és továbbadásáért felel (ez tehát a belépőpont a kernelbe). Sajnos nem látható a jelen helyzetben, hogy milyen függvények hívódnak meg a kernelben. Ehhez arra lenne szükség, hogy a debugger ismerje a kernel debug információit, amihez viszont szükséges lenne a kernel újrafordítása.
- 1.2, 5. pont, végtelen ciklus: A kernelmodulok nagyon magas prioritáson futnak. Jóval magasabban, mint például az ablakozórendszer. Ilyenkor tehát a rendszer „él“, de a felhasználó felé semmilyen jelzést nem tud adni, hiszen minden futásidőt a modul visz el.
- 1.2, 6. pont, kernel panic: Van különbség. Ilyenkor a kernel felé azt jelezzük, hogy a rendszer magja instabil állapotba került, így azt le kell állítani. Ilyenkor tehát ténylegesen megáll a kernel futása és nem csak az ablakozórendszer áll le. Ez a függvényhívás tipikusan olyan, amit egy felhasználói módú alkalmazás nem tudna meghívni.