Operating Systems - part 2

# Lab: Processes, Threads and system calls in Linux

P. Geens – P. Philippaerts – R.Swennen

Assignment:

- Start by reading this document thoroughly before you start!
- Make sure you understand all aspects of this lab. (**note: you will need to consult external resources**)
- You will work for two consecutive lessons on this lab in groups of 2 students (**larger groups are not allowed**). During the last lab session there is a short skills test. This test contains similar questions to the ones asked in the lab, including the extra ex.
  Answers to the test questions contain at least solutions (counts for 1/3) and their explanations (counts for 2/3, where relevant).
- Internet access is not allowed during the test.
- Linux one-liners must only output exactly what is asked, nothing more, nothing less. An example will be given by the lecturer.
- Because a part of the last class is occupied by the test and/or demonstration it is **necessary** that the lab is largely finished by the beginning of the 2nd lab.
- Read the document 'checklist voor het indienen van taken', available on I:\gt\shared\communicatie

## *0. Prerequisites:*

Thorough knowledge of the Linux shell techniques and concepts from the courses Operating Systems 1 and Computernetworks 2 are necessary in order manage this lab. It is the responsibility of the student to catch up the missing knowledge quickly.

## *1. The /proc filesystem:*

Since version 0.99.x of the Linux kernel the `/proc` file system has been introduced. It is a virtual file system that the represents the current state of the kernel as a collection of directories and files. In this lab we will examine the data available in `/proc` to improve our understanding on operating systems internals.

The `/proc` filesystem contains virtual files and directories with data about available processes in memory and the state/condition of the kernel. The `/proc` filesystem resides only in memory, not on disk. This way it can be changed quickly. Because it is a representation of the actual state of processes and kernel, it's generally not needed to save it's contents. This special filesystem is sometimes called a pseudo filesystem.

In fact it is impossible to save every change in the condition of the operating system to a file, this would have a negative impact on disk performance, and because this writing to the disk itself would result in different operating system state changes. E.g. the amount of context switches is kept in `/proc/stat` amongst many other things. If this would be an actual file on disk, every context switch would result in an update of this file, but every write generates a context switch by itself...

The intention is to represent data about the operating system and all it's processes in a simple and structured way. This information can now be easily read in files, whereas before you needed complicated system calls to obtain the needed info. Some of this information is easily readable while others seem nearly impossible to comprehend. Handy tools like ps, free, ntp, lspci, ... will make this information more human readable.
Most of the files are only readable, but some will also permit writing, and thus one could change the behavior of the running kernel. Most of the files in `/proc/sys` can be changed.

e.g. `echo myhostname > /proc/sys/kernel/hostname`

Give a detailed listing of the contents of the `/proc` directory:

```
pigee@:~$ ls -l /proc
dr-xr-xr-x    3 root     root             0 Aug 10 20:08 1
dr-xr-xr-x    3 pigee    pigee            0 Aug 10 20:08 1032
dr-xr-xr-x    3 pigee    pigee            0 Aug 10 20:08 1035
dr-xr-xr-x    3 pigee    pigee            0 Aug 10 20:08 1039
dr-xr-xr-x    3 pigee    pigee            0 Aug 10 20:08 1041
dr-xr-xr-x    3 pigee    pigee            0 Aug 10 20:08 1054
dr-xr-xr-x    3 pigee    pigee            0 Aug 10 20:08 1057
…
-r--r--r--    1 root      root            0 Aug 10 20:08 stat
-r--r--r--    1 root      root              0 Aug 10 20:08 swaps
dr-xr-xr-x   11 root      root              0 Aug 10 20:08 sys
dr-xr-xr-x    2 root      root              0 Aug 10 20:08 sysvipc
dr-xr-xr-x    4 root      root              0 Aug 10 20:08 tty
-r--r--r--    1 root      root              0 Aug 10 20:08 uptime
-r--r--r--    1 root      root              0 Aug 10 20:08 version
```

Note the contents can be slightly different for various versions of the Linux kernel, and do depend of the actual condition of the system at the exact moment you list the contents.

E.g. check the files `version`, `cpuinfo` and `pci`. They contain information about the version of the Linux kernel, the processor and hardware plugged into the pci bus. The file `filesystems` contains all filesystems the kernel can manage.

## *2. Example: Information about processes:*

For every process in memory a subdirectory in /proc is created and named as the process ID, containing all information about that particular process. One of the processes which will always be there is process init with ID 1, owned by root. Init is responsible for starting the entire operating system. As you can see there are lots of processes running.

Choose a process to examine more closely. Preferably a process which you own, so you have sufficient rights to read everything. Note the process you use to watch and observe in `/proc` can be reached through `/proc/<pid_of_process>` or `/proc/self`.

Give a listing of all files in the /proc/PID directory representing your process. Here is a list with all files you should be able to see. (this can be slightly different depending on the kernel version)

- `cmdline`: command used to start the process, with all parameters used (empty for a zombie process)
- `cpu`: the cpu on which the process is executed now and in the previous burst
- `cwd`: link to the directory in which the process works
- `environ`: environmental variables for this process
- `exe`: link to the executable file which executes this process
- `fd`: contains symbolic links to the corresponding file descriptors
- `maps`: memory translation to libraries and programs
- `mem`: used memory (this file is not user readable)

- `mounts`: mounted filesystems
- `root`: a link to the root directory of the process
- `stat`: status of the process (quite unreadable)
- `statm`: memory usage of the process, from left to right:
  - total size (kB)
  - size of the memory portions (kB)
  - # shared pages
  - # pages of code
  - # pages of data
  - # pages of libraries
  - # dirty pages
- `status`: state of the process in a more readable format (also contains data from stat and statm)

Start with the file `/proc/<pid_of_process>/status`. This one is easily readable. Look at this file, it should look approximately like this:

```
Name:   mozilla-thunder
State:  S (sleeping)
Tgid:   1481
Pid:    1481
PPid:   1453
TracerPid:      0
Uid:    1000    1000    1000    1000
Gid:    1000    1000    1000    1000
FDSize: 256
Groups: 1000 20 24 29 1012
VmSize:    88916 kB
VmLck:         0 kB
VmRSS:     40736 kB
VmData:    46296 kB
VmStk:        76 kB
VmExe:        40 kB
VmLib:     30396 kB
SigPnd: 0000000000000000
SigBlk: 0000000080000000
SigIgn: 0000000020001000
SigCgt: 000000038001442f
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
```

### *3. Example: Information about threads:*

Threads can be implemented in various ways. In user-space, in kernel-space, or both (like Solaris). Linux thread handling is exactly the same as process handling, and are implemented as kernel-space threads so they can easily be scheduled on multiple processors.

In `/proc` we can not only find information about certain processes but also about possible threads from that particular process. Look at the example below.

We look for the PID of slapd (multithreaded ldap daemon) with some help of the command ps:

```
pigee@:~$ ps aux | grep slapd | grep -v grep
root     25212  0.7  3.2 538908 108616 ?     Ss   Mar05  34:46
/usr/sbin/slapd …
```

Threads in Linux are analogue to processes and thus information about threads can be obtained from files in the `/proc` filesystem just like we can for regular processes. In `/proc/<PID>/task/`

we can find all threads related to a particular process. Just like real processes every thread has it's own unique identifier (PID).

```
pigee@:~$  ls /proc/25212/
auxv  cmdline  cwd  environ  exe  fd  maps  mem  mounts  oom_adj
oom_score  root  seccomp  stat  statm  status  task  wchan
```

The process itself is also in the task list:

```
pigee@:~$  ls /proc/25212/task/
14481  14735  19247  21698  22197  22198  23029  25212  25213  25214
25242  25334  28132  4752  5783  5788  5789  5790
```

```
pigee@:~$  ls /proc/25212/task/14481
auxv  cmdline  cwd  environ  exe  fd  maps  mem  mounts  oom_adj
oom_score  root  seccomp  stat  statm  status  wchan
```

A more thorough explanation about the `/proc` filesystem can be found in the manual pages of `proc(5)` and various web pages. You will need them to be able to solve the exercises.

## *4. Exercises on the `/proc` filesystem:*

- ***Some exercises can only be solved by using techniques described in the following pages!***
- *Your answer should contain both the actual command (oneliner), and a detailed explanation!*
- *Information should be gathered manually by parsing relevant `/proc` files. System commands like `ps` may only be used for verification!*

Look for the right information in /proc and find an answer for the following questions using a suitable oneliner:

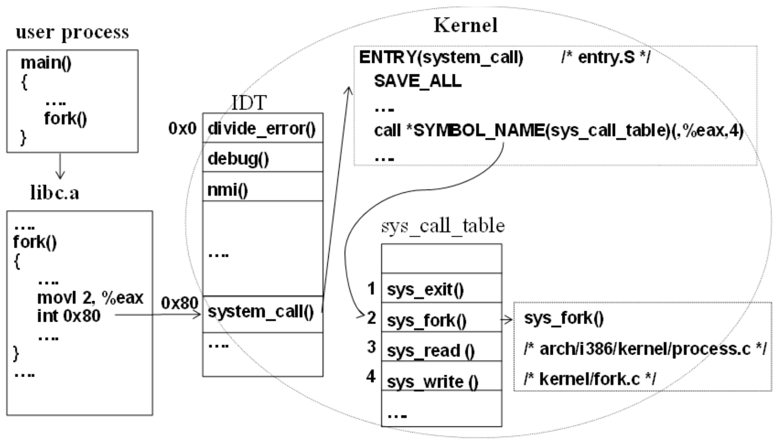
- Which processor does this machine have?
- How many context switches have occurred yet? And how many processes and interrupts? Is the number of context switches higher than the number of processes? Explain thoroughly! And why is the number of interrupts even bigger on some systems and smaller on others?
- How many non maskable interrupts have occurred yet? What does this mean?
- What type of interrupt has occurred the most since the start of our operating system?
- Where does the command `free` gets its data from? Interpret the output. i.e. calculate it's output from it's source.
- Is it possible that on a running system the output of the command `free` in the free column is the same for the first 2 rows? How can you force this?
- How much time was the processor unemployed since boot? Calculate the percentage of the total time which the processor was unemployed, and the percentage of the useful processor time that has been used by the operating system.
- How much memory is available on this machine, and how much is still available? Is the swap space used at the moment? Which process occupies the largest amount of system ram?
- What's in `/proc/kcore`, how big is this file?
- Find your neighbors mac address. Hint: ping him first.
- Which kernel modules are loaded?
- How can you (as `root)` through /proc:
    - Change the reaction of the system to ctrl-alt-del? What are the possible actions?
    - Don't respond to ICMP Echo requests anymore (ping requests)?
- List (use only one command line) all processes in the state RUNNING.

## *5. System Call's:*

Processes in the linux operating system can run in either user mode or kernel mode. Privileged instructions, like access to io devices are only possible when running in kernel mode. User mode processes can have access to these privileged instructions through the use of system call's. System call's are very much line ordinary procedure call's. They provide an interface between user programs and the kernel. Every system call has it's own specific use.

In `/usr/include/asm/unistd_32.h` and `/usr/include/asm/unistd_64.h` you will find a list of all supported linux system call's. How much unique call's are available?

On [http://www.cheat-sheets.org/saved-copy/Linux_Syscall_quickref.pdf](http://www.cheat-sheets.org/saved-copy/Linux_Syscall_quickref.pdf) you can find a short description of every system call. Under linux these system routines are implemented by an interrupt (int 0x80). This interrupt transfers the control to the kernel. The following figure is an example of the `fork()` system call:




## *6. Tracing System Call's with `strace`:*

Using the command *`strace`* a system administrator is able to see every system call a process is executing and whether they were successful or not/.

`Strace` is very easy to use. Consider the following example, a string will be printed on the terminal screen.

```
pigee@:~$ echo “strace demo”
strace demo
```

Even for a very basic command like the one above, many system call's, and thus switches to kernel mode are needed:

```
pigee@:~$ strace echo "strace demo"
execve("/bin/echo", ["echo", "strace demo"], [/* 20 vars */]) = 0
uname({sys="Linux", node="Mordor", ...}) = 0
brk(0)                                  = 0x804d000
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40017000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=10601, ...}) = 0
old_mmap(NULL, 10601, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
```

```
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`Z\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1254468, ...}) = 0
old_mmap(NULL, 1264780, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x4001b000
old_mmap(0x40145000, 36864, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x129000) = 0x40145000
old_mmap(0x4014e000, 7308, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4014e000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40150000
set_thread_area({entry_number:-1 -> 6, base_addr:0x401502a0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40018000, 10601)               = 0
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=290576, ...}) = 0
mmap2(NULL, 290576, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40151000
close(3)                                = 0
brk(0)                                  = 0x804d000
brk(0x806e000)                          = 0x806e000
brk(0)                                  = 0x806e000
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40198000
write(1, "strace demo\n", 12strace demo
)           = 12
munmap(0x40198000, 4096)                = 0
exit_group(0)                           = ?
```

## *7. Some exercises on system call's:*

*- Your answer should contain both the actual command used, and a detailed explanation!*

Use the program strace to find the answer to the following questions: read the relevant manpages carefully for more information. Make sure your 'oneliner' only shows whats asked, nothing more and nothing less.

- How many system call's do we need to read a file with the command `cat`?
- Which system call consumed the most time in previous example?
- Look at the system call's of a process that's already running. What is the process doing?
- The command `ps aux` gets its information out of `/proc`. How many files were accessed in `/proc` during this operation?

## *8. More advance tracing using `DTrace` or `SystemTap`.*

DTrace is a comprehensive dynamic tracing framework created by Sun Microsystems for troubleshooting kernel and application problems on production systems in real time. Originally developed for Solaris, it has since been released under the free Common Development and Distribution License (CDDL) and has been ported to several other Unix-like systems.

DTrace can be used to get a global overview of a running system, such as the amount of memory, CPU time, filesystem and network resources used by the active processes. It can also provide much more fine-grained information, such as a log of the arguments with which a specific function is being called, or a list of the processes accessing a specific file.

http://en.wikipedia.org/wiki/DTrace

SystemTap provides free software (GPL) infrastructure to simplify the gathering of information about the running Linux system. This assists diagnosis of a performance or functional problem. SystemTap eliminates the need for the developer to go through the tedious and disruptive instrument, recompile, install, and reboot sequence that may be otherwise required to collect data.

SystemTap provides a simple command line interface and scripting language for writing instrumentation for a live running kernel. We are publishing samples, as well as enlarging the internal "tapset" script library to aid reuse and abstraction.

Among other tracing/probing tools, SystemTap is the tool of choice for complex tasks that may require live analysis, programmable on-line response, and whole-system symbolic access. SystemTap can also handle simple tracing jobs.

Current project members include Red Hat, IBM, Hitachi, and Oracle.

http://sourceware.org/systemtap/

### *9. Simple Solaris DTrace examples:*

List, and count all system call's over over all processes:

```
pigee@:~$ dtrace -n 'syscall:::entry { @num[probefunc] = count(); }'
dtrace: description 'syscall:::entry ' matched 333 probes
^C
  unlink                                                            1
  wait4                                                             1
  access                                                            2
  getuid32                                                          2
  waitpid                                                           2
  chmod                                                             4
  fdatasync                                                         4
  inotify_add_watch                                                 4
  munmap                                                            9
  stat64                                                           10
  fstat64                                                          12
  mmap2                                                            13
  nanosleep                                                        13
  rt_sigaction                                                     14
  socketcall                                                       14
  open                                                             17
  rt_sigprocmask                                                   21
  sigreturn                                                        24
  close                                                            27
  fcntl64                                                          43
  _llseek                                                          59
  writev                                                           59
  ioctl                                                            71
  _newselect                                                       76
  setitimer                                                        99
  sched_yield                                                     154
  poll                                                            201
  futex                                                           275
  time                                                            426
  gettimeofday                                                    622
  clock_gettime                                                  1540
  write                                                          6310
```

```
  read                                                          6405
```

As you can see, in this case the most used system call's were `read()` and `write()`.

Same example, but now including the process name:

```
pigee@:~$ dtrace -n 'syscall:::entry { @num[probefunc, execname] = count(); }'
dtrace: description 'syscall:::entry ' matched 333 probes
^C

  newselect                        gdl_box                            1
  brk                              dtrace                             1
  clone                            smbd                               1
  close                            devkit-disks-da                    1
  ...
  ioctl                            gnome-panel                        2
  ioctl                            gnome-settings-                    2
  ...
  poll                             gnome-panel                        5
  poll                             quasselcore                        5
  ...
  writev                           Xorg                              20
  poll                             /usr/bin/termin                   23
  poll                             soffice.bin                       23
  read                             soffice.bin                       23
  setgroups32                      smbd                              28
  socketcall                       smbd                              29
  _newselect                       Xorg                              30
  gettimeofday                     /usr/bin/termin                   32
  setregid32                       smbd                              33
  setreuid32                       smbd                              33
  fcntl64                          quasselcore                       34
  read                             Xorg                              48
  gettimeofday                     smbd                              52
  write                            quasselcore                       53
  setitimer                        Xorg                              54
  _llseek                          quasselcore                       62
  fcntl64                          smbd                              66
  getegid32                        smbd                              75
  geteuid32                        smbd                              76
  gettimeofday                     soffice.bin                       85
  clock_gettime                    Xorg                             112
  time                             dtrace                           245
  futex                            dtrace                           249
  read                             /usr/bin/termin                  444
  lseek                            /usr/bin/termin                  528
  clock_gettime                    dtrace                          1227
  gettimeofday                     Xorg                           23460
```

## *10. Some extra exercises (similar questions will be asked during the test / demonstration):*

- The file /proc/<PID>/status contains among several other things the corresponding name of the PID and the number of threads. Create a one-liner which produces the following output:

  ```
  33 nscd 4348
  27 apache2 28054
  27 apache2 28051
  ```

```
3 slapd 4272
1 watchdog/0 5
1 udevd 2478
1 syslogd 4219
1 su 15147
```

From left to right: number of threads, process name, process identifier.
So you need to print a process top 10 sorted by the amount of threads. Suppress all error messages!

- With the lsof command you can retrieve a list of open files. e.g. `lsof -i tcp` prints out all open files which correspond with tcp connections. Without the necessary rights, this command will not give you any output. Find out which files from /proc the command `lsof -i tcp` is trying to open, but fails because of insufficient permissions. Almost all these files belong to a particular process. (I.e. opened by this process)
  Provide a list of the command that started these processes. You can find this in `/proc/<pid>/cmdline`. Output: `<pid>: command`, according to the sample output below (no empty lines) As you can see with Apache, processes may be started more than once.
  Sample output:

```
1: /sbin/init
2380: /usr/sbin/apache2-kstart
2381: /usr/sbin/apache2-kstart
2382: /usr/sbin/apache2-kstart
2437: /sbin/udevd--daemon
2900: sshd: root@pts/0
2902: -bash
2934: su-pigee
...
4520: sshd: root@pts/1
4522: -bash
27190: /usr/sbin/apache2-kstart
```

- Consider you were running a program which analyzes your hard drive, looking for 'stuff'. It takes about 24 hours to finish. Unfortunately, after 23 hours you accidentally closed your ssh session. After logging in again you see that the process is still running. You forgot to append `| tee -a logfile.log` so you can't get to the results. How could you use strace to solve this problem, i.e. show the remaining output of the process running in another bash session.

  You could simulate this with a tail -f /tmp/somefile in one ssh session and then use your strace magic to read the output in another session.