Process and File Watching and Monitoring Utilities

- Linux provides a wide variety of powerful and easy-to-use system API calls and utilities to monitor and watch file and process activity.
- Such utilities were obviously designed to enable the development of applications that are useful for system administrators and analysts vis-à-vis performance and security.
- However, just as with any utility the use of such tools can be subverted for use in eavesdropping, exfiltration, and backdoors.
- We will examine three types of mechanisms:
    - The much-maligned (and deprecated) ***dnotify*** application
    - The **inotify** suite of API calls that has replaced **dnotify**
    - Examining the process table using **"/proc**"

Using "dnotify"

- ***dnotify*** (**d**irectory **notify**) is a very simple system call that makes it possible to execute a command every time the contents of a specific directory under watch undergoes changes.
- It is executed from the command line with the following syntax:

  dnotify [*OPTION*]... *DIRECTORY*... [-e *COMMAND*...]
- The above invocation will execute COMMAND every time the contents of the specified DIRECTORY change. If a command is not specified, 'echo {}' is assumed. The string '{}' in the command specification is replaced by the name of the directory that was updated.
- The exact change that will trigger the COMMAND to be executed is defined by OPTION. A change is determined by the '--access', '--modify', '--create', '--delete', '--rename' and '--attrib' options.
- These options may be combined. If none of them are specified, create and delete are assumed:

  -A, --access

  Trigger the command when a file in the specified directory is accessed (read).

  -M, --modify

  Trigger the command when a file in the specified directory is modified (i.e. written or truncated).

  -C, --create

  Trigger the command when a new file is created in the directory.

-D, --delete

Trigger the command when a file is removed from the directory.

-R, --rename

Trigger the command when a file in the specified directory is renamed.

-B, --attrib

Trigger the command when a file in the specified directory has its attributes changed (as a result of chmod or chown).

-a, --all

Trigger the command when any one of the events above occurs. That is, this is a shorthand for '-AMCDRO'.

-e, --execute=*COMMAND*...

This option specifies the command to execute. All arguments (including possible options) following this option are treated as arguments to the command.

-f, --file=*FILE*

The option specifies a file containing directory names to monitor, as if they had been specified on the command line. There must be one directory name per line in the file.

This option can be used multiple times.

-p, --processes=*COUNT*

This option specifies the maximum number of commands to be run at the same time. It is possible to allow an unlimited number of commands to be run at the same time by specifying COUNT as 0 or -1.

The default value is 1.

-q, --queue=*DEPTH*

This option specifies the maximum number of commands to queue up while there are other commands already executing. This is usually used in conjunction with the --processes option. An unlimited depth of the queue can be achieved by specifying DEPTH as -1.

The default value is -1.

-t, --times=*COUNT*

Exit dnotify once the command has been run (and finished running) COUNT times.

-o, --once

Exit dnotify when the specified command has been run once. This is equivalent to '--times 1'.

-r, --recursive

Traverse the specified directories and monitor all subdirectories (at any depth) as well.

dnotify will only scan directories once to find out subdirectories to monitor when recursive mode is enabled. If you want to monitor any subdirectory created, you will have to stop and restart dnotify every time a create or delete event occurs.

-b, --background

Run dnotify in the background - detach from terminal. Prior to executing any commands, dnotify will change the current directory to the root directory ('/'). Relative path names of directories to monitor will be made absolute automaticly, but make sure the command you specify does not rely on the current directory being unchanged.

-s, --silent, --quiet

Do not print warnings when the executed commands exits with a non-zero return value.

--help

Show summary of options.

--version

Output version information and exit.

Examples

- The following example will echo 'change' to stdout every time a file (or part of file) in /etc is read.
  dnotify -A /etc -e echo change

- Execute the script 'informdelete' every time a file in /var/mail is created or deleted.
  dnotify -CD /var/mail -e informdelete

- The following is a very simple program that shows how to execute dnotify from within a C program:

```
#include  <stdio.h>
#include  <stdlib.h>

int main (void)
{
        system ("dnotify -s -C /temp -e /email-scripts/exd&");
        exit (0);
}
```

- Now, if a new file is created in “/temp”, the script “exd” will be executed. The following is an arbitrary sample script:

```
echo "foobar" | mail -s "New File" foo@bar.com
```

- Although dnotify is simple to configure and use, there are numerous drawbacks to this application:
    - dnotify requires the opening of one file descriptor per each directory that is to be put under a watch.
    - The file descriptor pins the directory, disallowing the backing device to be unmounted, which does not work very well with removable media.
    - Watching many directories results in many open file descriptors, possibly hitting a per-process descriptor limit.
    - dnotify is directory-based. You only learn about changes to directories. Thus making it very difficult to track file changes within a directory.
    - dnotify uses signals to communicate with user-space, specifically it uses SIGIO. This makes it very difficult and cumbersome to queue events.

Using "inotify"

- **inotify** (inode notify) is a file system event-monitoring mechanism that is designed to serve as an effective replacement for dnotify, which was the de facto file-monitoring mechanism supported in older kernels.
- It is a powerful, fine-grained, asynchronous mechanism ideally suited for a variety of file-monitoring needs including security and performance.
- This is a kernel-based file change notification system that allows applications to request the monitoring of a set of files against a list of events.
- When the event occurs, the application is notified. To be useful, such a feature must be simple to use, lightweight with little overhead and flexible.
- It should be easy to add new watches and receive notification of events.
- This is a timely replacement for dnotify. The following are some of highlights of this framework:
    - inotify's interface is a device node, not the SIGIO signal.
    - One a single file descriptor need be opened to the device node. This eliminates the pinning of directories or opening a multiple file descriptors.
    - The basic usage is simple: open the device, issue simple commands via ioctl(), and then block on the device. It returns events when there are events to be returned.
    - The ***select()*** call can be used on the device node.
    - ***inotify*** provides an event that generates an alert when the files system under watch is unmounted.
    - ***inotify*** can watch directories or files. More specifically, it is **inode-based** (**i**node **notify**).

- Events can be monitored using **inotify-tools**, which provides a set system calls that are invoked from command line.

- Our main focus is to use the API to develop monitoring code and applications. However, the following are some examples of command-line invocation:

- **inotifywait**
    - This command simply blocks for inotify events, making it appropriate for use in shell scripts. It can watch any set of files and directories, and can recursively watch entire directory trees.

Example 1

```
#!/bin/sh
# A slightly complex but actually useful example
inotifywait -mrq --timefmt '%d/%m/%y %H:%M' --format '%T %f' \
 -e close_write /home/foo | while read date time file; do
   rsync /home/foo/${file} rsync://foo@bar.com/backup/${file} && \
   echo "At ${time} on ${date}, file ${file} was backed up via rsync"
done
```

- This may be the most efficient way to block for changes on files from a shell script.
- If a specific event to be caught is not specified, all events will be caught, and the particular event that occurred is printed to stdout.

Example 2

```
#!/bin/sh
EVENT=$(inotifywait --format '%e' ~/file1)
[ $? != 0 ] && exit
[ "$EVENT" = "MODIFY" ] && echo 'file modified!'
[ "$EVENT" = "DELETE_SELF" ] && echo 'file deleted!'
# etc...
```

- **Inotifywatch**
    - ***inotifywatch*** collects filesystem usage statistics and outputs counts of each inotify event.

Example 1

The following example shows how to recursively watch **/data/temp** for 60 seconds. At the end of the 60 second period, all the vent statistics are printed out:

```
[root@milliways ~]# inotifywatch -v -e access -e modify -t 60 -r /data/temp
Establishing watches...
Setting up watch(es) on /data/temp
OK, /data/temp is now being watched.
Total of 1 watches.
Finished establishing watches, now collecting statistics.
Will listen for events for 60 seconds.
total  access  modify  filename
6      2       4       /data/temp/
```

```
[root@milliways ~]#
```

Initializing indotify

- As mentioned earlier, our main focus is to use the API calls within a C program to monitor events. The first step is to initialize inotify. The ***inotify_init()*** system call instantiates an inotify instance inside the kernel and returns the associated file descriptor.
- The following code fragment illustrates the use of this system call:

```
int fd;

fd = inotify_init ();
if (fd < 0)
        perror ("inotify_init");
```

- The core of inotify is the watch, which consists of a pathname specifying what to watch and an event mask specifying what to watch for.
- inotify can watch for many different events: opens, closes, reads, writes, creates, deletes, moves, metadata changes and unmounts.
- Each inotify instance can have thousands of watches, each watch for a different list of events.
- This is a kernel-based file change notification system that allows applications to request the monitoring of a set of files against a list of events.
- Watches Watches are added with the ***inotify_add_watch()*** system call:

```
int inotify_add_watch (int fd, const char *path, __u32 mask);
```

- A call to **inotify_add_watch()** adds a watch for the one or more events given by the bitmask mask on the file path to the inotify instance associated with the file descriptor fd.
- A call to inotify_add_watch() adds a watch for the one or more events given by the bitmask mask on the file path to the inotify instance associated with the file descriptor fd.
- On success, the call returns a watch descriptor, which is used to identify this particular watch uniquely. On failure, minus one is returned and errno is set as appropriate.
- On success, the call returns a watch descriptor, which is used to identify this particular watch uniquely. On failure, minus one is returned and errno is set as appropriate.

- The following code fragment illustrates its usage:

```
int wd;

wd = inotify_add_watch (fd, “/data/temp”,IN_MODIFY | IN_CREATE | IN_DELETE);

if (wd < 0)
        perror ("inotify_add_watch");
```

- This example adds a watch on the directory /data/temp for any modifications, file creations or file deletions.
- The following table shows all the valid events.

Event	Description
IN_ACCESS	File was read from.
IN_MODIFY	File was written to.
IN_ATTRIB	File's metadata (inode or xattr) was changed.
IN_CLOSE_WRITE	File was closed (and was open for writing).
IN_CLOSE_NOWRITE	File was closed (and was not open for writing).
IN_OPEN	File was opened.
IN_MOVED_FROM	File was moved away from watch.
IN_MOVED_TO	File was moved to watch.
IN_DELETE	File was deleted.
IN_DELETE_SELF	The watch itself was deleted.

Receiving Events

- With inotify initialized and watches added, the nest step is start receiving and processing events.
- Events are queued asynchronously, in real time as the events happen, but they are read synchronously via the ***read()*** system call. The call **blocks until events are ready** and then returns all available events once any event is queued.
- Events are delivered in the form of an **inotify_event** structure, which is defined as:

```
struct inotify_event
{
 __s32 wd;            /* watch descriptor */
     __u32 mask;      /* watch mask */
     __u32 cookie;    /* cookie to synchronize two events */
     __u32 len;       /* length (including nulls) of name */
     char name[0];    /* stub for possible name */
};
```

- The **wd** field is the watch descriptor originally returned by inotify_add_watch(). The application is responsible for mapping this identifier back to the filename.
- The **mask** field is a bitmask representing the event that occurred.
- The **cookie** field is a unique identifier linking together two related but separate events. It is used to link together an IN_MOVED_FROM and an IN_MOVED_TO event. We will look at it later.
- The **len** field is the length of the name field or nonzero if this event does not have a name. The length contains any potential padding—that is, the result of strlen() on the name field may be smaller than len.
- The **name** field contains the name of the object to which the event occurred, relative to wd, if applicable.
- For example, if a watch for writes in /data triggers an event on the writing to /home/foo, the name field will contain foo, and the wd field will link back to the /data watch.
- Conversely, in watching the file /etc/fstab for reads, a triggered read event will have a len of zero and no associated name whatsoever, because the watch descriptor associates directly with the affected file.
- Because the name field is dynamic, the size of the buffer passed to read() is unknown. If the size is too small, the system call returns zero, alerting the application. inotify, however, allows user space to “slurp” multiple events at once.
- Consequently, most applications should pass in a large buffer, which inotify will fill with as many events as possible.

- The following code fragment illustrates this concept:

```
/* size of the event structure, not counting name */
#define EVENT_SIZE  (sizeof (struct inotify_event))

/* reasonable guess as to size of 1024 events */
#define BUF_LEN        (1024 * (EVENT_SIZE + 16)

char buf[BUF_LEN];
int len, i = 0;

len = read (fd, buf, BUF_LEN);
if (len < 0) {
      if (errno == EINTR)
            /* need to reissue system call */
      else
            perror ("read");
} else if (!len)
      /* BUF_LEN too small? */

while (i < len) {
      struct inotify_event *event;

      event = (struct inotify_event *) &buf[i];

      printf ("wd=%d mask=%u cookie=%u len=%u\n",  event->wd, event->mask, event-
>cookie, event->len);

      if (event->len)
            printf ("name=%s\n", event->name);

      i += EVENT_SIZE + event->len;
}
```

Using *select()* to read events

- Having the main process block on a read() system call is very inefficient unless the application is heavily threaded.

- A much better approach is to use **epoll**, **poll** or **select** on the inotify file descriptor, thus allowing inotify to be multiplexed along with other I/O and optionally integrated into an application's mainloop.

- The following is an example of monitoring the inotify file descriptor with select():

```
struct timeval time;
fd_set rfds;
int ret;

/* timeout after five seconds */
time.tv_sec = 10;
time.tv_usec = 0;

/* zero-out the fd_set */
FD_ZERO (&rfds);

/*
 * add the inotify fd to the fd_set -- of course,
 * your application will probably want to add
 * other file descriptors here, too
 */
FD_SET (fd, &rfds);

ret = select (fd + 1, &rfds, NULL, NULL, &time);
if (ret < 0)
        perror ("select");
else if (!ret)
        /* timed out! */
else if (FD_ISSET (fd, &rfds)
        /* inotify events are available! */
```

Modifying Watches

- A watch is modified by calling **inotify_add_watch()** with an updated event mask. If the watch already exists, the mask is simply updated and the original watch descriptor is returned.

Removing Watches

- Watches are removed with the **inotify_rm_watch()** system call:

  int inotify_rm_watch (int fd, int wd);

- A call to inotify_rm_watch() removes the watch associated with the watch descriptor wd from the inotify instance associated with the file descriptor fd.
- The call returns zero on success and negative one on failure, in which case errno is set as appropriate.

- The following code fragment illustrates its use:

```
int ret;

ret = inotify_rm_watch (fd, wd);
if (ret)
        perror ("inotify_rm_watch");
```

Shutting inotify Down

- To destroy any existing watches, pending events and the inotify instance itself, invoke the close() system call on the inotify instance's file descriptor.

- For example:

```
int ret;

ret = close (fd);
if (ret)
        perror ("close");
```

Handling an unmount

- One of the main criticisms of dnotify is that a dnotify watch on a directory requires that directory to remain open.

- Consequently, watching a directory on, say, a USB drive prevents the drive from being unmounted. inotify solves this problem by not requiring that any file be open.
- inotify takes this one step further, though, and sends out the IN_UNMOUNT event when the filesystem on which a file resides is unmounted. It also automatically destroys the watch and cleanup.

Handling Moves

- Move events are complicated because inotify may be watching the directory that the file is moved to or from, but not the other.
- Because of this, it is not always possible to alert the user of the source and destination of a file involved in a move. inotify is able to alert the application to both only if the application is watching both directories.
- In that case, inotify emits an IN_MOVED_FROM from the watch descriptor of the source directory, and it emits an IN_MOVED_TO from the watch descriptor of the destination directory. If watching only one or the other, only the one event will be sent.
- To tie together two disparate moved to/from events, inotify sets the cookie field in the inotify_event structure to a unique nonzero value.
- Two events with matching cookies are thus related, one showing the source and one showing the destination of the move.

Obtaining the Size of the Queue

- The size of the pending event queue can be obtained via FIONREAD:

```
unsigned int queue_len;
int ret;

ret = ioctl (fd, FIONREAD, &queue_len);
if (ret < 0)
      perror ("ioctl");
else
      printf ("%u bytes pending in queue\n", queue_len);
```

- This is useful to implement throttling: reading from the queue only when the number of events has grown sufficiently large.

Configuring inotify

- inotify is configurable via ***procfs*** and ***sysctl***.
- **/proc/sys/filesystem/inotify/max_queued_events** is the maximum number of events that can be queued at once. If the queue reaches this size, new events are dropped, but the IN_Q_OVERFLOW event is always sent.
- With a significantly large queue, overflows are rare even if watching many objects. The default value is 16,384 events per queue.
- **/proc/sys/filesystem/inotify/max_user_instances** is the maximum number of inotify instances that a given user can instantiate. The default value is 128 instances, per user.
- **/proc/sys/filesystem/inotify/max_user_watches** is the maximum number of watches per instance. The default value is 8,192 watches, per instance.
- The example provided illustrates the use of all the essential inotify monitoring calls.

Process Monitoring and Searching

- Many applications (both security and performance related) require that certain processes be running at all times and this makes it necessary to implement code within critical applications that checks on such processes.
- All of the kernel and process information is maintained in the /proc directory.
- The proc filesystem is a pseudo-filesystem rooted at **/proc** that contains user-accessible objects that pertain to the runtime state of the kernel and, by extension, the executing processes that run on top of it.
- **/proc** is very special in that it is also a virtual filesystem. It's sometimes referred to as a process information pseudo-file system. It doesn't contain 'real' files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc). For this reason it can be regarded as a control and information centre for the kernel.
- The term "pseudo" is used because the proc filesystem exists only as a reflection of the in-memory kernel data structures it displays. This is why most files and directories within /proc are 0 bytes in size.
- A listing of this directory will produce a listing similar to the following:

```
total 0
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 10
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1020
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 11
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1344
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1345
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 136
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 137
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1373
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1374
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1375
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 139
dr-xr-xr-x  6 root      root             0 2008-05-16 16:43 14063
dr-xr-xr-x  6 root      root             0 2008-05-16 16:47 14072
dr-xr-xr-x  6 root      root             0 2008-05-16 16:48 14075
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 144
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 147
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1737
dr-xr-xr-x  6 rpc       root             0 2008-05-15 11:00 1747
dr-xr-xr-x  6 rpcuser   rpcuser          0 2008-05-15 11:00 1767
……..
……..

dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 1990
dr-xr-xr-x  6 root      root             0 2008-05-15 11:00 2
dr-xr-xr-x  6 ntp       ntp              0 2008-05-15 11:00 2005
dr-xr-xr-x  6 cyrus     mail             0 2008-05-15 11:00 2136
……..
```

- Each of the numbered directories corresponds to an actual process ID. Looking at the process table, we can match processes with the associated process ID.

- For example, the process table might indicate the following for the secure shell server:

```
#ps auxw | grep sshd
root     1990  0.0  0.0   6080  1048 ?      Ss   May15   0:01 /usr/sbin/sshd
```

- More details for this process can be obtained by looking at the associated files in the directory for this process, **/proc/1990**.

- Note that the entry has a file size of 0. The file doesn't actually contain any data; it just acts as a pointer to where the actual process information resides.

- For example, a listing of the files in the **/proc/1990** directory produces the following output:

```
total 0
dr-xr-xr-x 2 root root 0 2008-05-19 13:55 attr
-r-------- 1 root root 0 2008-05-19 13:55 auxv
-r--r--r-- 1 root root 0 2008-05-19 13:55 cgroup
--w------- 1 root root 0 2008-05-19 13:55 clear_refs
-r--r--r-- 1 root root 0 2008-05-15 11:00 cmdline
-rw-r--r-- 1 root root 0 2008-05-19 13:55 coredump_filter
-r--r--r-- 1 root root 0 2008-05-19 13:55 cpuset
lrwxrwxrwx 1 root root 0 2008-05-19 13:55 cwd -> /
-r-------- 1 root root 0 2008-05-19 13:55 environ
lrwxrwxrwx 1 root root 0 2008-05-15 11:00 exe -> /usr/sbin/sshd
dr-x------ 2 root root 0 2008-05-19 13:55 fd
dr-x------ 2 root root 0 2008-05-19 13:55 fdinfo
-r--r--r-- 1 root root 0 2008-05-19 13:55 io
-r-------- 1 root root 0 2008-05-19 13:55 limits
-rw-r--r-- 1 root root 0 2008-05-19 13:55 loginuid
-r-------- 1 root root 0 2008-05-19 13:55 maps
-rw------- 1 root root 0 2008-05-19 13:55 mem
-r--r--r-- 1 root root 0 2008-05-19 13:55 mounts
-r-------- 1 root root 0 2008-05-19 13:55 mountstats
-rw-r--r-- 1 root root 0 2008-05-19 13:55 oom_adj
-r--r--r-- 1 root root 0 2008-05-19 13:55 oom_score
lrwxrwxrwx 1 root root 0 2008-05-19 13:55 root -> /
-rw-r--r-- 1 root root 0 2008-05-19 13:55 sched
-r--r--r-- 1 root root 0 2008-05-19 13:55 schedstat
-r-------- 1 root root 0 2008-05-19 13:55 smaps
-r--r--r-- 1 root root 0 2008-05-15 11:00 stat
-r--r--r-- 1 root root 0 2008-05-19 13:55 statm
-r--r--r-- 1 root root 0 2008-05-15 11:00 status
dr-xr-xr-x 3 root root 0 2008-05-19 13:55 task
-r--r--r-- 1 root root 0 2008-05-19 13:55 wchan
```

- The purpose and contents of some of the main items in these files is as follows:
    - /proc/1990/cmdline

      Command line arguments.
    - /proc/1990/cpu

      Current and last cpu in which it was executed.
    - /proc/1990/cwd

      Link to the current working directory.
    - /proc/1990/environ

      Values of environment variables.
    - /proc/1990/exe

      Link to the executable of this process.
    - /proc/1990/fd

      Directory, which contains all file descriptors.
    - /proc/1990/maps

      Memory maps to executables (shared objects) and library files.
    - /proc/1990/mem

      Memory held by this process.
    - /proc/1990/root

      Link to the root directory of this process.
    - /proc/1990/stat

      Process status.
    - /proc/1990/statm

      Process memory status information.
    - /proc/1990/status

      Process status in human readable form.

- We can examine the contents of each file in the /proc/1990 directory (the zero size notwithstanding) as follows:

```
#cat status

Name: sshd
State:     S (sleeping)
Tgid: 1990
Pid:  1990
PPid: 1
TracerPid:  0
Uid:  0     0     0     0
Gid:  0     0     0     0
FDSize:    32
Groups:
VmPeak:         6084 kB
VmSize:         6080 kB
VmLck:             0 kB
VmHWM:          1048 kB
VmRSS:          1048 kB
VmData:          400 kB
VmStk:            84 kB
VmExe:           388 kB
VmLib:          4960 kB
VmPTE:            20 kB
StaBrk:    b7ff5000 kB
Brk:  b84a4000 kB
StaStk:    bff546d0 kB
ExecLim:   b7fee000
Threads:   1
SigQ: 1/32760
SigPnd:    0000000000000000
ShdPnd:    0000000000000000
SigBlk:    0000000000000000
SigIgn:    0000000000001000
SigCgt:    0000000180014005
CapInh:    0000000000000000
CapPrm:    00000000ffffffff
CapEff:    00000000ffffffff
Cpus_allowed:    00000003
Mems_allowed:    1
voluntary_ctxt_switches:    12617
nonvoluntary_ctxt_switches:  4571
```

- The files in the /proc directory act very similar to the process ID subdirectory files. For example, examining the contents of the /proc/interrupts file displays something like the following:

```
           CPU0       CPU1
  0:        250          2   IO-APIC-edge      timer
  1:       8924         32   IO-APIC-edge      i8042
  3:          1          0   IO-APIC-edge
  4:          1          1   IO-APIC-edge
  6:          3          2   IO-APIC-edge      floppy
  7:          0          0   IO-APIC-edge      parport0
  8:          0          1   IO-APIC-edge      rtc
  9:          1          0   IO-APIC-fasteoi   acpi
 12:     230835        120   IO-APIC-edge      i8042
 14:    3933638        103   IO-APIC-edge      libata
 15:    5744381         68   IO-APIC-edge      libata
 16:       1194         39   IO-APIC-fasteoi   ohci_hcd:usb1
 17:     216685       5426   IO-APIC-fasteoi   aic79xx
 18:        338        149   IO-APIC-fasteoi   aic79xx
 20:     854051         32   IO-APIC-fasteoi   eth0
 21:          0          0   IO-APIC-fasteoi   radeon@pci:0000:01:05.0
NMI:          0          0   Non-maskable interrupts
LOC:   12117597   10650920   Local timer interrupts
RES:     397029    1809500   Rescheduling interrupts
CAL:        184        156   function call interrupts
TLB:       5889      24863   TLB shootdowns
TRM:          0          0   Thermal event interrupts
SPU:          0          0   Spurious interrupts
ERR:          0
MIS:          0
```

- Another very interesting directory to examine is **/proc/net**. This provides a huge amount of information realted to the issues we have been discussing in this course.

- Broadly speaking, getting any information about a process is simply and exercise in going through the /proc directory and searching (many string comparisons) through all subdirectories and files for the required information.

- The example provided illustrates the mechanism and the series of systems calls involved in a simple search for the existence of a process and obtaining its PID.