National University of Computer and Emerging Sciences

Operating System Lab - 10

Lab Manual

Contents

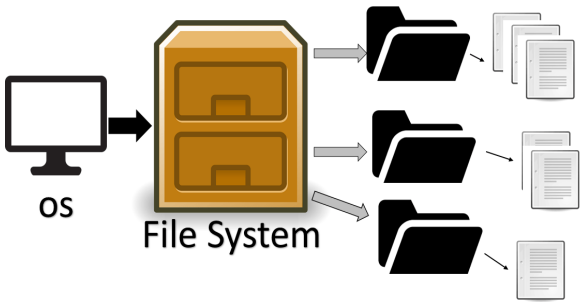
Objective

The purpose of this lab is:

- Creating your own proc file
- Viewing file in lsmod and /proc
- Introducing to sequence files
    - Creating sequence files
    - The file operations struct and file related functions
- Creating a sequence
    - The sequence operations struct and sequence related functions
- Printing a sequence of even numbers in a sequence file

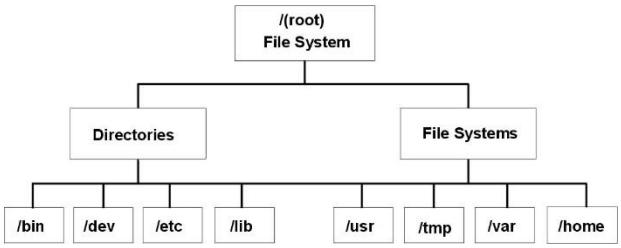
What is a File System?

The data unit to save an item (datum) in your computer is a file. To manage your data properly as per their relevance and importance you need a file system. The general definition of a file system is:



"*A file system is the way in which files are named and where they are placed logically for storage and retrieval. The DOS, Windows, OS/2, Macintosh, and UNIX-based operating systems all have file systems in which files are placed somewhere in a hierarchical (tree) structure.*

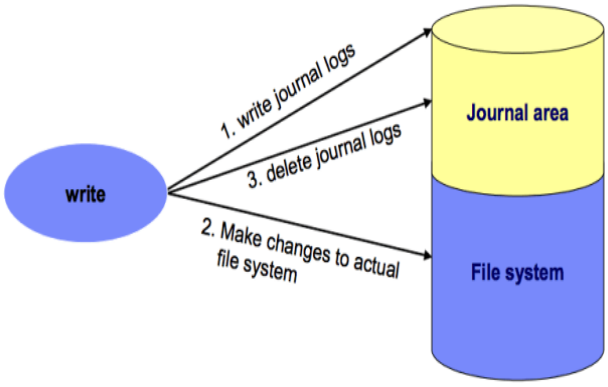
Sometimes the term refers to the part of an operating system or an added-on program that supports a file system as defined above. Examples of such add-on file systems include the Network File System (NFS) and the Andrew file system (AFS)."



Reference: http://searchstorage.techtarget.com/definition/file-system

What is Journaling?

A journaling file system is more reliable when it comes to data storage. Journaling file systems do not necessarily prevent corruption, but they do prevent inconsistency and are much faster at file system checks than non-journaled file systems. If a power failure happens while you are saving a file, the save will not complete and you end up with corrupted data and an inconsistent file system. Instead of actually writing directly to the part of the disk where the file is stored, it first writes it to another part of the hard drive and notes the necessary changes to a log, then in the background it goes through each entry to the journal and begins to complete the task, and when the task is complete, it checks it off on the list. Thus the file system is always in a consistent state (the file got saved, the journal reports it as not completely saved, or the



journal is inconsistent (but can be rebuilt from the file system)). Some journaling file systems can prevent corruption as well by writing data twice.

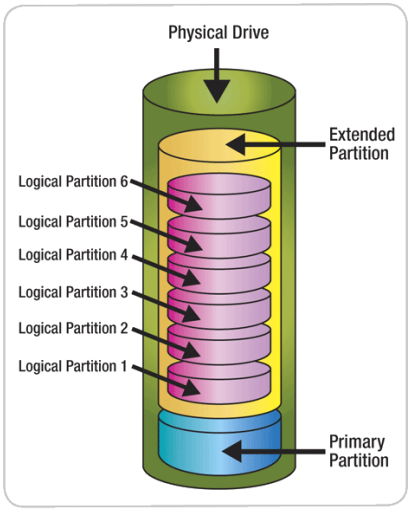
File System Comparison

File System	Max File Size	Max Partition Size	Journaling	Notes
FAT16	2 GB	2 GB	No	Legacy
FAT32	4 GB	8 TB	No	Legacy
exFAT	16 EB	128 PB	Yes	New file system against FAT32
NTFS	2 TB	256 TB	No	(For Windows Compatibility) NTFS-3g is installed by default in Ubuntu, allowing Read/Write support
ext2	2 TB	32 TB	Yes	Legacy
ext3	2 TB	32 TB	Yes	Standard linux filesystem for many years. Best choice for super-standard installation.
ext4	16 TB	1 EB	Yes	Modern iteration of ext3. Best choice for new installations where super-standard isn't necessary.
reiserFS	8 TB	16 TB	Yes	No longer well-maintained.
JFS	4 PB	32 PB	Yes	Created by IBM - Not well maintained.
XFS	8 EB	8 EB	Yes	Created by SGI. Best choice for a mix of stability and advanced journaling.

GB = Gigabyte (1024 MB) | TB = Terabyte (1024 GB) | PB = Petabyte (1024 TB) | EB = Exabyte (1024 PB)

Above you'll see a brief comparison of two main attributes of different filesystems, the max file size and the largest a partition of that data can be.

What is Partitioning?



Usually partitions refer to the physical disks partitions (primary, logical and extended), but it may seem strange that Linux uses more than one partition on the same disk, even when using the standard installation procedure, so some explanation is called for.

One of the goals of having different partitions is to achieve higher data security in case of disaster. By dividing the hard disk in partitions, data can be grouped and separated. When an accident occurs, only the data in the partition that got the hit will be damaged, while the data on the other partitions will most likely survive.

A simple example: a user creates a script, a program or a web application that starts filling up the disk. If the disk contains only one big partition, the entire system will stop functioning if the disk is full. If the user stores the data on a

separate partition, then only that (data) partition will be affected, while the system partitions and possible other data partitions keep functioning.

There are two kinds of major partitions on a Linux system:

- **data partition:** normal Linux system data, including the root partition containing all the data to start up and run the system.
- **swap partition**: expansion of the computer's physical memory, extra memory on hard disk.

Swap space (indicated with swap) is only accessible for the system itself, and is hidden from view during normal operation. Swap is the system that ensures, like on normal UNIX systems, that you can keep on working, whatever happens. On Linux, you will virtually never see irritating messages like Out of memory, please close some applications first and try again, because of this extra memory. The swap or virtual memory procedure has long been adopted by operating systems outside the UNIX world by now.

Using memory on a hard disk is naturally slower than using the real memory chips of a computer, but having this little extra is a great comfort.

The kernel is on a separate partition as well in many distributions, because it is the most important file of your system. If this is the case, you will find that you also have a /boot partition, holding your kernel(s) and accompanying data files.

Once the partitions are made, you can only add more. Changing sizes or properties of existing partitions is possible but not advisable.

Every Partition has its own File System

By imagining all those file systems together, we can form an idea of the tree-structure of the entire system, but it is not as simple as that. In a file system, a file is represented by an inode, a kind of serial number containing information about the actual data that makes up the file: to whom this file belongs, and where it located on the hard disk, this information is usually saved as the file header.

Every partition has its own set of inodes; throughout a system with multiple partitions, files with the same inode number can exist. Since the address on different partitions may be same, so does the owner of the file.

Each inode describes a data structure on the hard disk, storing the properties of a file, including the physical location of the file data. When a hard disk is initialized to accept data storage, usually during the initial system installation process or when adding extra disks to an existing system, a fixed number of inodes per partition is created. This number will be the maximum amount of files, of all types (including directories, special files, links etc.) that can exist at the same time on the partition. We typically count on having 1 inode per 2 to 8 kilobytes of storage.

At the time a new file is created, it gets a free inode. In that inode is the following information:

- Owner and group owner of the file.
- File type (regular, directory, ...)
- Permissions on the file
- Date and time of creation, last read and change.

- Date and time this information has been changed in the inode.
- Number of links to this file (referrals or shortcuts)
- File size
- An address defining the actual location of the file data.

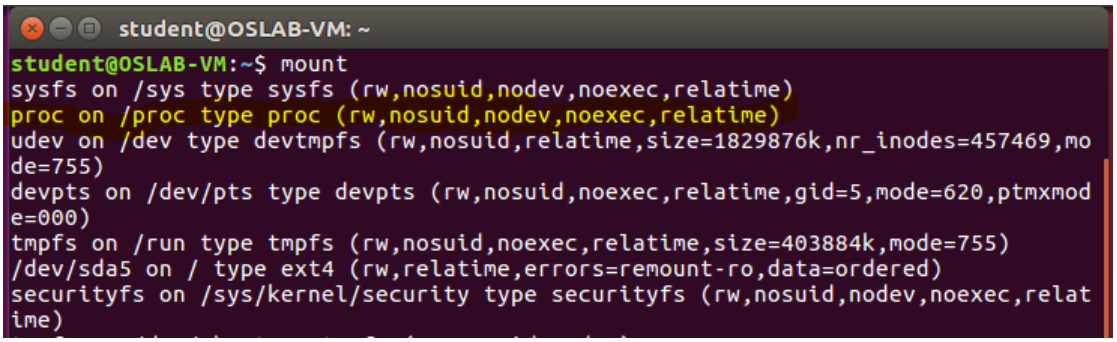
The only information not included in an inode, is the file name and directory. These are stored in the special directory files. By comparing file names and inode numbers, the system can make up a tree-structure that the user understands. Users can display inode numbers using the -i option to ls. The inodes have their own separate space on the disk.

The /proc File System

"The /proc/ directory — also called the proc file system — contains a hierarchy of special files which represent the current state of the kernel — allowing applications and users to peer into the kernel's view of the system. Under Linux, all data are stored as files. Most users are familiar with the two primary types of files: text and binary. But the /proc/ directory contains another type of file called a virtual file. It is for this reason that /proc/ is often referred to as a virtual file system. These virtual files have unique qualities. Most of them are listed as zero bytes in size and yet when one is viewed, it can contain a large amount of information. In addition, most of the time and date settings on virtual files reflect the current time and date, indicative of the fact they are constantly updated. Virtual files such as /proc/interrupts, /proc/meminfo, /proc/mounts, and /proc/partitions provide an up-to-the-moment glimpse of the system's hardware. Others, like the /proc/ file systems file and the /proc/sys/ directory provide system configuration information and interfaces."

Reference: http://www.centos.org/docs/5/html/Deployment_Guide-en-US/ch-proc.html

The proof that the /proc is a file system not just a directory is that it is mounted on the system, just as other file systems on other partitions. You can view them by writing:



The C codes that dynamically create virtual files are not modules, they are rather built into kernel explicitly, they are not buildable as a module, and they are not loadable as a service. The Makefile that jointly compile /proc files is: (you can view it at */lib/modules/4.13.0-38-generic/build/fs/proc/Makefile*)

```
#
# Makefile for the Linux proc filesystem routines.
#
obj-y += proc.o
proc-y := nommu.o task_nommu.o
proc-$(CONFIG_MMU) := task_mmu.o
```

```
proc-y += inode.o root.o base.o generic.o array.o \
fd.o
proc-$(CONFIG_TTY) += proc_tty.o
proc-y += cmdline.o
proc-y += consoles.o
proc-y += cpuinfo.o
proc-y += devices.o
proc-y += interrupts.o
proc-y += loadavg.o
proc-y += meminfo.o
proc-y += stat.o
proc-y += uptime.o
proc-y += version.o
proc-y += softirqs.o
proc-y += namespaces.o
proc-y += self.o
proc-$(CONFIG_PROC_SYSCTL) += proc_sysctl.o
proc-$(CONFIG_NET) += proc_net.o
proc-$(CONFIG_PROC_KCORE) += kcore.o
proc-$(CONFIG_PROC_VMCORE) += vmcore.o
proc-$(CONFIG_PROC_DEVICETREE) += proc_devtree.o
proc-$(CONFIG_PRINTK) += kmsg.o
proc-$(CONFIG_PROC_PAGE_MONITOR) += page.o
```

This is a comprehensive Makefile that is compiling several proc files at once. The files named as xyz.o are the object files corresponding to their C source files. They don't have many parameters to configure, as a matter of fact you configured them initially when you configured your kernel. If you notice, you would see that some proc files are getting their configurations from system variables that you configured during Kernel configuration process, these variables are written before the object file names in brackets. Other proc files that don't have that system variable in front of them, are non – configurable i.e. they have no flexible option to be configured, they are represented as proc-y in above.

The Content of /proc Directory

Files in /proc

- **/proc/buddyinfo ->** This file is used primarily for diagnosing memory fragmentation issues.
- **/proc/cmdline ->**This file shows the parameters passed to the kernel at the time it is started.
- **/proc/cpuinfo ->** This virtual file identifies the type of processor used by your system
- **/proc/crypto ->** This file lists all installed cryptographic ciphers used by the Linux kernel, including additional details for each.
- **/proc/devices ->** This file displays the various character and block devices currently configured
- **/proc/filesystems ->** This file displays a list of the file system types currently supported by the kernel.

- **/proc/interrupts** -> This file records the number of interrupts per IRQ on the x86 architecture.
- **/proc/iomem** -> This file shows you the current map of the system's memory for each physical device
- **/proc/ioports** -> The output of /proc/ioports provides a list of currently registered port regions used for input or output communication with a device.
- **/proc/kcore** -> This file represents the physical memory of the system and is stored in the core file format.
- **/proc/kmsg** -> This file is used to hold messages generated by the kernel.
- **/proc/meminfo** -> This is one of the more commonly used files in the /proc/ directory, as it reports a large amount of valuable information about the systems RAM usage.
- **/proc/misc** -> This file lists miscellaneous drivers registered on the miscellaneous major device, which is device number 10:
- **/proc/modules** -> This file displays a list of all modules loaded into the kernel.
- **/proc/mounts** -> This file provides a list of all mounts in use by the system
- **/proc/partitions** -> This file contains partition block allocation information.
- **/proc/swaps** -> This file measures swap space and its utilization.
- **/proc/uptime** -> This file contains information detailing how long the system has been on since its last restart
- **/proc/version** -> This file specifies the version of the Linux kernel and gcc in use, as well as the version of Red Hat Enterprise Linux installed on the system

Directories in /proc

- **/proc/self/** -> The /proc/self/ directory is a link to the currently running process. This allows a process to look at itself without having to know its process ID.
- **/proc/bus/** -> This directory contains information specific to the various buses available on the system.
- **/proc/driver/** -> This directory contains information for specific drivers in use by the kernel.
- **/proc/fs/** -> This directory shows which file systems are exported.
- **/proc/ide/** -> This directory contains information about IDE devices on the system.
- **/proc/net/** -> This directory provides a comprehensive look at various networking parameters and statistics. Each directory and virtual file within this directory describes aspects of the system's network configuration.
- **/proc/sys/** -> The /proc/sys/ directory is different from others in /proc/ because it not only provides information about the system but also allows the system administrator to

immediately enable and disable kernel features.
Reference: https://www.centos.org/docs/5/html/Deployment_Guide-en-US/s1-proc-topfiles.html

After all, why on earth is /proc's behavior so different?

You may have heard that "Appearances may be deceiving", so is the case for the /proc directory. The /proc directory is actually not just a directory, it's a whole file system that dynamically creates sequence files on the run time, generate its contents and removes them as user stops viewing them.

The Sequence Files

Proc files have severe limitation that if their output exceeds a single page in RAM, it simply stops working. This is because the file is created in RAM and if it constitutes more than one page in RAM, the operating system would have to create a Process Control Block and process page table for it, then they would no more be memory efficient. Sequence files emerged as a solution to this problem. It protects against overflow of the output buffer and easily handles procfs files that are larger than one page. It also provides methods for traversing a list of kernel items and iterating on that list. The sequence file is also prone to the same problem that if OUTPUT SIZE exceeds the buffer size (size of page in RAM) then it would have to stop, you need to implement some method to restart them immediately from the same location they stopped. The main idea or improvement that sequence files brought was that they generated output in parts not the whole file at once as the proc files used to do. The output is now generated in the memory in the form of a linked list that is traversed by the sequence files, generating output step by step, this decreases the size of output, and so reduces the chances of interruption due to buffer overflow.

In short, seq_file operates by using "pull" methods, pulling or asking for data from seq_file operations methods, whereas the previous procfs methods pushed data into output buffers.

Viewing Files in /proc

You can view any virtual proc file using any editor or just simply writing cat, as below:

```
Cat /proc/version
```

Reference: https://elixir.bootlin.com/linux/v4.1.3/source/fs/proc

Below is the code of version.c

```
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/utsname.h>
static int version_proc_show(struct seq_file *m, void *v)
{
seq_printf(m, linux_proc_banner,
utsname()->sysname,
utsname()->release,
utsname()->version);
return 0;
}
```

```
static int version_proc_open(struct inode *inode, struct file *file)
{
return single_open(file, version_proc_show, NULL);
}
static const struct file_operations version_proc_fops = {
.open = version_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = single_release,
};
static int __init proc_version_init(void)
{
proc_create("version", 0, NULL, &version_proc_fops);
return 0;
}
fs_initcall(proc_version_init);
```

This one is simplest among all proc files, and reveals the quick recipe to create virtual files. To create a proc entry/ virtual file, you need to do the following:

1. Create a proc entry of whatever name you like
2. Create a file operations structure that would define the file operations on the virtual file that would be created dynamically
3. Create an open method for this file that would in return call a method to generate the file contents
4. Create a method that would generate the file contents

Preparations to write a proc file generator code:

You need the following header files explicitly added to your code:

```
fs.h
proc_fs.h
seq_file.h
```

write the function call for fs_initcall(loading_function_name), this does the same task as that of module_init(), remember this is not a module, therefore there are a few changes:

1. module_init is replaced by fs_initcall (for newer versions of Linux)
2. There is no module_exit, because this file system is never supposed to exit

Step I: The Loading of the File

First of all you need to create an initialize or loading function just to create the proc file entry in /proc with your desired name.

```
static int __init proc_version_init(void)
{
proc_create("version", 0, NULL, &version_proc_fops);
return 0;
```

```
}
```

The proc_create function is defined in proc_fs.h. This function above is a loading function as marked via __init. The parameters for proc_create are:

1. name of entry to be created in /proc directory
2. user rights on the file (0 means default)
3. NULL – name of parent directory under /proc, version is directly under proc so has no parent directory
4. file operations structure for the virtual file behind the entry

Step II: The File Operations Structure

Now you need to define operations on file, so that for each interrupt that comes for the file, the Operating system would know what to do.

```
static const struct file_operations version_proc_fops = {
.open = version_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = single_release,
};
```

This structure has some mandatory components such as open, read, write and seek. You need to define operations for these file operations. Here read and seek are set to default values of user rights on the file.

Step III: The Open Method for the Virtual File

Now that we have defined all the values to default for the virtual file, only task that needs to be accomplished is to write the open method for the virtual file.

```
static int version_proc_open(struct inode *inode, struct file *file)
{
return single_open(file, version_proc_show, NULL);
}
```

As described above in file systems, each file is represented by a file pointer that contains file data and an inode pointer that contains file meta data (data about the file OR file header). Since our file has no peculiar file description, we don't alter the value of inode, just open the file given to us by the file pointer and generating its output using the second argument i.e. a function to generate file data. Since we are not saving this file onto the hard drive the path for the file is NULL (the third parameter). You need not to do anything else here.

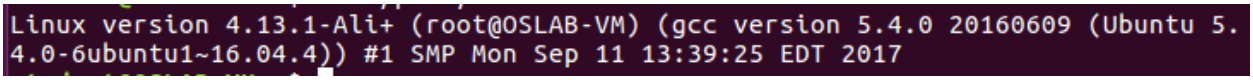
Step IV: The Data Generator Method for Virtual File

After the file has been opened we need to generate the data for it. This task is performed using the following code snippet:

```
static int version_proc_show(struct seq_file *m, void *v)
```

```
{
seq_printf(m, linux_proc_banner,
utsname()->sysname,
utsname()->release,
utsname()->version);
return 0;
}
```

As evident from the code, this code generates four values on the file. The output file looks like the following:



The seq_printf is used to output on sequence file, the parameters format is as follows

1. File pointer to produce data
2. Data to print

Linux_proc_banner = prints the banner of Linux proc i.e . `Linux version`

`Utsname() -> sysname = 4.13.1-Ali+`

`Utsname() -> release = root@OSLAB-VM`

`Utsname() -> version = (gcc version 5.4.0 (Ubuntu 5.4.0-ubuntu1~16.04.4))`
`#1 SMP Mon Sep 11 13:39:25 EDT 2017`

The function of seq_printf varies from context to context, as you will see in future it works very close to the printf function, and it would be really very handy for you to work with.
Since this sequence file has a very little output to generate, the whole output has been generated at once. As we will see later that the sequence files with large amounts of outputs have a different show function.

Putting this Altogether

When this file starts execution the operating system finds out the line fs_initcall() just as it searches for module_init() in a module. From there it knows the location of loading function (remember the function's name is its pointer in code segment). It then immediately goes to the loading function and creates the proc entry. It then sets in waiting till user views the file. If user asks the Kernel to show that file, it allocates an inode pointer and a file pointer in RAM and passes it to the file opening method specified in file_operations structure. From there it knows the location of the function that would generate the contents of the file. Now it allocates a temporary space of type void for the content generation method to work on, and passes the file pointer to it. At this point the file is opened and is empty, now the version_proc_show function produces the output on the file using seq_printf.

Creating Your Own /proc File

Now we would follow the same steps describe above to create our own proc file with the following minor differences:

1. Since we have not yet embedded our code into kernel, we must write module_exit so as to give the module a safe way to exit.
2. Due to the same reason we need to write a module instead of a C code similar to the one above.

So, we now make our own proc file using Loadable Kernel Module, the approach we have been using so in these operating systems labs. Our primary task is to reuse those four steps described earlier in our module. The final product looks something like this:

```
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/jiffies.h>
static int jiffies_proc_show(struct seq_file *m, void *v)
{
seq_printf(m, "%llu\n", (unsigned long long) get_jiffies_64());
return 0;
}
static int jiffies_proc_open(struct inode *inode, struct file *file)
{
return single_open(file, jiffies_proc_show, NULL);
}
static const struct file_operations jiffies_proc_fops = {
.owner = THIS_MODULE,
.open = jiffies_proc_open,
.read = seq_read,
.llseek = seq_lseek,
.release = single_release,
};
static int __init jiffies_proc_init(void)
{
proc_create("mod8_jiffies", 0, NULL, &jiffies_proc_fops);
return 0;
}
static void __exit jiffies_proc_exit(void)
{
remove_proc_entry("mod8_jiffies", NULL);
}
module_init(jiffies_proc_init);
module_exit(jiffies_proc_exit);
MODULE_AUTHOR("ABC");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("A jiffies /proc file");
```

Here everything is almost the same, just the two of them are worth explaining:

1 – the show function of this module

The show function of this module is used to show the value of an exported kernel symbol

JIFFIES, a counter that keeps track of number of system ticks and is based on the frequency of system timer (normally the tick rate is around 110 Hz).

```
static int jiffies_proc_show(struct seq_file *m, void *v)
{
seq_printf(m, "%llu\n", (unsigned long long) get_jiffies_64());
return 0;
}
```

As you may have studied accessor and mutator functions of classes in the Computer programming course, the function get_jiffies_64() is doing almost the same thing, encapsulating the original variable jiffies in its original location and providing you the interface by this function. By default the *unsigned long* is 32 bits long, we need a value that is of 64 bits, so we used *long long unsigned* (%llu) to print the value of jiffies in sequence file.

2 – the unloading function of this module

Well, by default it is our responsibility to free or undo everything that is done while loading before unloading the module, so in this function we just removed the /proc entry created in loading function

```
static void __exit jiffies_proc_exit(void)
{
remove_proc_entry("mod8_jiffies", NULL);
}
```

Remove_proc_entry contains two parameters

- What to remove
- Where to find it under proc tree (NULL means direct decendent)

We now run the module using insmod, and viola the /proc shows our entry named as mod8_jiffies as you can see below:

```
root@OSLAB-VM:/home/student/Desktop/jiffies# ls /proc
1     1108  1468  175   2024  2114  220   2319  2582  34   4875  53   96        dma          kpagecgroup  partitions    uptime
10    1109  15    18    2028  2120  2212  2328  2596  37   49    54   97        driver       kpagecount   sched_debug   version
100   1110  1548  182   2032  2123  2231  2338  26    38   4959  55   98        execdomains  kpageflags   schedstat     vmallocinfo
101   1113  1564  19    2036  2124  2237  2340  2648  386  4999  6    99        fb           loadavg      scsi          vmstat
1022  1163  1574  1940  2041  2127  2239  2372  268   39   50    6086 acpi      filesystems  locks        self          zoneinfo
1026  1171  16    1953  2059  2129  2240  2384  27    4    5012  6140 asound    fs           mdstat       slabinfo
1027  1173  1679  1955  2065  2130  2241  24    272   40   5030  6145 buddyinfo interrupts   meminfo      softirqs
1029  12    1680  1979  2067  2131  2260  2417  2791  41   5064  6147 bus       iomem        misc         stat
1032  1214  169   1981  2082  2133  2261  2476  28    42   511   7    cgroups   ioports      mod8_jiffies swaps
1063  1253  170   1985  2083  2141  2277  2483  2940  43   5170  8    cmdline   irq          modules      sys
1066  128   172   1988  2088  217   2280  2490  2986  44   5220  887  consoles  kallsyms     mounts       sysrq-trigger
107   13    1720  2     2091  219   2281  2494  30    45   5239  9    cpuinfo   kcore        mpt          sysvipc
1072  14    173   20    21    2190  2291  25    3004  46   5246  911  crypto    keys         mtrr         thread-self
1082  1453  174   2010  2107  2191  2293  251   31    47   5257  931  devices   key-users    net          timer_list
11    1459  1746  2015  2112  22    2309  2575  32    48   5258  957  diskstats kmsg         pagetypeinfo tty
root@OSLAB-VM:/home/student/Desktop/jiffies#
```

And by ordering kernel, we can view this file as follows:

```
cat /proc/mod8_jiffies
4295193005
```

Using Sequence File to Generate Large Amount of Output

So far, our sequence files were merely printing just a single value in virtual file, what about the files producing a large amount of output. Well they use a structure that is backbone of the sequence files, the SEQUENCE.

What is a sequence?

A sequence, in literal means the list of elements which can be as simple as the natural numbers, even numbers, and prime numbers to as complicated as the list of devices installed on the system, list of processes waiting in queue to be served by processor etc.

The sequence files were an improvement over the traditional proc files because they generated output in the form of bits and pieces to avoid the buffer over flow (page size exceeded) limitation. The output is generated in the form of iterations that print all elements in the file. The contents of file are stored in the Kernel at different locations that are either collected to form an array or they are kept in their places just linked to each other to form a linked list. No matter how programmer arranges the file data, it is the duty of programmer to provide address and offset to each element in the list (sequence) till we finish with it.

The contents list here are maintained using the sequences defined in seq_file.h in the form of a structure written as:

```
struct seq_operations {
void * (*start) (struct seq_file *m, loff_t *pos);
void (*stop) (struct seq_file *m, void *v);
void * (*next) (struct seq_file *m, void *v, loff_t *pos);
int (*show) (struct seq_file *m, void *v);
};
```

The sequence uses the following two system's defined attributes to access elements in the list:

- **void *v** -> it represents the address of the current element in the list, by default address type is void*, since data at each element is not pre-decided, the pointer is not of any particular data type, so you need to type cast if you want to access the value at that address
- **loff_t *pos** -> it represents the loop variable, the iterator that is initialized by system to zero, but the increment is duty of programmer.

As, it is clear from above the functionality provided by these methods:

- **start** -> makes any necessary allocations and assignments as needed by the sequence, and returns the address to the first element in the list
- **stop** -> frees any assignments and allocations on finishing the task by checking the value of v
- **next** -> this function takes the responsibility of setting the value of v and pos to the next element in the list
- **show** -> prints the current element in the list

Take a look on the following code, as it generates the sequence of even number in a proc file using iterations to print.

Overview of the Program

This is an extremely simple example of generating sequence via sequences in sequence files, it generates even numbers up to a defined limit 10. You can change it when inserting the module as explained in labs. If you change it to a larger value, say 1000, this is what it shows in messages log:

```
... Entering start(), pos = 0.
... In start(), even_ptr = ffff880126662b88.
... In show(), even = 0.
... snip ...
... In show(), even = 186.
... In next(), v = ffff880126662b88, pos = 93.
... In show(), even = 188.
... Entering stop().
... v is ffff880126662b88.
... In stop(), even_ptr = ffff880126662b88.
... Freeing and clearing even_ptr.
... Entering start(), pos = 94.
... In start(), even_ptr = ffff880126662b88.
... In show(), even = 188.
...
```

Did you see something weird? The program entered stop when it was not even near to finishing the job, and then suddenly it starts again, why so?

Since the sequence files still face the limitations of size of output, if the output file (the virtual sequence file) size exceeds the page size, the sequence is stopped and the void * v is emptied/ freed i.e. stop function is called. Now it is programmer's responsibility to handle such events and restart the sequence if the task is left unfinished, but the question arises here is that how a module knows that the task was interrupted, not finished, since it has already freed the pointer of data element, v?

Here loff_t *pos comes into spot light, since it is the only variable that is not flushed during the restarting process, you can track the progress using the value at pos, but for that you need to update pos at each step when you update the value at v. Both of your start and stop functions should be designed in such a fashion to handle this situation. Now we explain the code of the module.

The code marked as grey above is the general module code, whereas the code written in black is the use of sequence to generate even numbers. Only new thing in the module code is following (the calling of sequence in opening the sequence file)

```
static int ct_open(struct inode *inode, struct file *file)
{
return seq_open(file, &ct_seq_ops);
};
```

This is the point where you call the sequence, to generate the output in file. Now we describe how we used the sequences:

1 – the seq_operations structure

Just like the file_operations structure, we define seq_operations structure to define operations on sequences.

```
static struct seq_operations ct_seq_ops = {
.start = ct_seq_start,
.next = ct_seq_next,
.stop = ct_seq_stop,
.show = ct_seq_show
};
```

This code above defines that we have written four functions to implement the four operations on sequences, the kernel should call them as per need.

2 – the seq_start function

This function does the constructor / loading job for the sequence and returns the address to the first or next element in the list. The code for that is below:

```
static void* ct_seq_start(struct seq_file *s, loff_t *pos)
{
printk(KERN_INFO "Entering start(), pos = %Ld.\n", *pos);
if ((*pos) >= limit) { // are we done?
printk(KERN_INFO "Apparently, we're done.\n");
return NULL;
}
// Allocate an integer to hold our increasing even value.
even_ptr = kmalloc(sizeof(int), GFP_KERNEL);
if (!even_ptr) // fatal kernel allocation error
return NULL;
printk(KERN_INFO "In start(), even_ptr = %pX.\n", even_ptr);
*even_ptr = (*pos) * 2;
return even_ptr;
}
```

Here initially you print the message to represent the current index / offset/ iteration value pos indicating that we are in start now. Then we check that if the offset exceeds limit, we don't need to do anything, its already all done, so we exit returning NULL.

If we have not yet finished up and this is either start or restart call, we allocate some space in kernel memory using kmalloc. The second parameter to kmalloc is the urgency of memory allocation requirement, the two options are:

- GFP_ATOMIC -> a rapid memory allocation option with high chance of failures
- GFP_KERNEL -> a slower option that seldom fails in allocation

If your memory allocation requirement is not very large, it is feasible to use kmalloc, use vmalloc otherwise. The next lines are for exception handling, that is if evenptr fails to get allocated, the start should return NULL, a sign of empty list of elements. If the allocation goes successful, the address pointed by evenptr is printed. Remember that the loff_t type pointer pos is the iterator that is initialized by system to zero, in case of restarting the pos will

represent the index from where to continue, so we set the value at evenptr (the even number to be generated) to be the twice of it. As expected we are now returning the address of first or next even number.

3 – the seq_show function

This function takes the responsibility to print in sequence file the current value pointed by v or evenptr so the code is:

```
static int ct_seq_show(struct seq_file *s, void *v)
{
printk(KERN_INFO "In show(), even = %d.\n", *((int*)v));
seq_printf(s, "The current value of the even number is %d\n",
*((int*)v));
return 0;
}
```

As expected the function prints the integer value at v represented as *(int *)v with type casting it to integer.

4 – the seq_next function

The next function in the sequence does a very important job of traversing the list. The exact working in words for seq_next is to increment the value of offset in pos and set the address of next element in v, so the code is:

```
static void* ct_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
int* val_ptr;
printk(KERN_INFO "In next(), v = %pX, pos = %Ld.\n", v, *pos);
(*pos)++; // increase my position counter
if ((*pos) >= limit) // are we done?
return NULL;
val_ptr = (int *) v; // address of current even value
(*val_ptr) += 2; // increase it by two
return v;
}
```

Since for each element in the list of even numbers we don't need to allocate separate memory space as they need not to be coexistent, so here is a shortcut we are taking is to avoiding navigation among different memory addresses, for each iteration we simply increment the value of pos and update the value at the address stored in v.

As you can see, first of all we printed a message to trace the presence of control in next function, next we immediately increment pos and check if the value exceeds the limit, if so, our list has ended up and we need to exit now, otherwise we use a temporary reference variable val_ptr to update the value currently stored at the address hold by v. Now that the pos is updated, the value at v is modified, the next method has finished its task, it is now ready to exit, so we return the untouched value of v to exit.

5 – the seq_stop function

This function is responsible for unloading, unregistering and setting free all the resources reserved during the sequence creation and maintenance. Remember that the evenptr and v are both pointer to the same resource.

```
static void ct_seq_stop(struct seq_file *s, void *v)
```

```
{
printk(KERN_INFO "Entering stop().\n");
if (v) {
printk(KERN_INFO "v is %pX.\n", v);
} else {
printk(KERN_INFO "v is null.\n");
}
printk(KERN_INFO "In stop(), even_ptr = %pX.\n", even_ptr);
if (even_ptr) {
printk(KERN_INFO "Freeing and clearing even_ptr.\n");
kfree(even_ptr);
even_ptr = NULL;
} else {
printk(KERN_INFO "even_ptr is already null.\n");
}
}
```

There is a four-step verification that the sequence has finished its task.

The first step is when the next function sets the value NULL to the pointer v representing that the list is finished.

In the second step, on determining that v has been set to NULL the Kernel is bound to call stop method. Since v is already free, we set the evenptrfree too, since we already know that stop is called means either the process output has exceeded the page size or it has finished its task, in either case the pointer to the next item evenptr it needs to be freed.

In third step the start is called, it cross checks if the sequence is finished, if it is it displays the message that “apparently we are done” and returns NULL indicating that there is no more elements in the list.

In fourth step again stop is called, this time evenptr is already free, so it just prints that evenptr is already null and finally exits.

After providing proof of presence in this function, we checked the value of v, if a sequence is not yet finished, the v will have some value other than NULL, then it prints the value, but if the function is already finished.

The output for the code in virtual file looks like below, plainly transparent to user the implementation behind.

```
cat /proc/evens
The current value of the even number is 0
The current value of the even number is 2
The current value of the even number is 4
The current value of the even number is 6
The current value of the even number is 8
The current value of the even number is 10
The current value of the even number is 12
The current value of the even number is 14
The current value of the even number is 16
The current value of the even number is 18
```

Well some of its implementation can be traced from the kernel messages log, which looks like the one below:

```
$dmesg
... Entering start(), pos = 0.
... In start(), even_ptr = ffff880126662c98.
... In show(), even = 0.
... In next(), v = ffff880126662c98, pos = 0.
... In show(), even = 2.
... snip ...
... Entering stop().
... v is null.
... In stop(), even_ptr = ffff880126662c98.
... Freeing and clearing even_ptr.
... Entering start(), pos = 10.
... Apparently, we're done.
... Entering stop().
... v is null.
... In stop(), even_ptr = (null).
... even_ptr is already null.
```

The code above below where the red code shows the first time execution of stop, the blue color is showing the third step, when start is called again. Shown in green is the final call to stop.