Patrick Scott
Paolo Milan
CSCE 311 Project 3 Report

1. Is /proc a file system? We used one of the functions in the second project (cat /proc/$pid/maps); please enumerate five other functions that you like most. In addition to peeking (i.e., reading) kernel secrets, can you also write to /proc? If so, please give one example.

2. What are the uses of Loadable Kernel Modules? Is it possible to implement Project 2 (i.e., reading through the user space of a given process) using LKM?

/proc is a special type of filesystem in Unix-like operating systems that presents information about processes and other system information in a hierarchical file-like structure, providing a more convenient and standardized method for dynamically accessing process data that is held in the kernel.

Five functions that we like are :

1. list_add - Adds a new entry after the specified head. This is good for implementing stacks.
2. strim - Removes leading and trailing whitespace from str
3. Memscan - Finds a character in an area of memory
4. devm_request_resource - Request and reserve an I/O or memory resource
5. reallocate_resource - allocate a slot in the resource tree given range & alignment. The resource will be relocated if the new size cannot be reallocated in the current location.

It is also possible to write in a /proc file. It works the same way as read, a function can be called when the /proc file is written. But to read it, it is a little different. Read usually comes from the user. In kernel space you can use a buffer to write. To do this you have to import data from user space to kernel space, with functions copy_from_user or get_user.

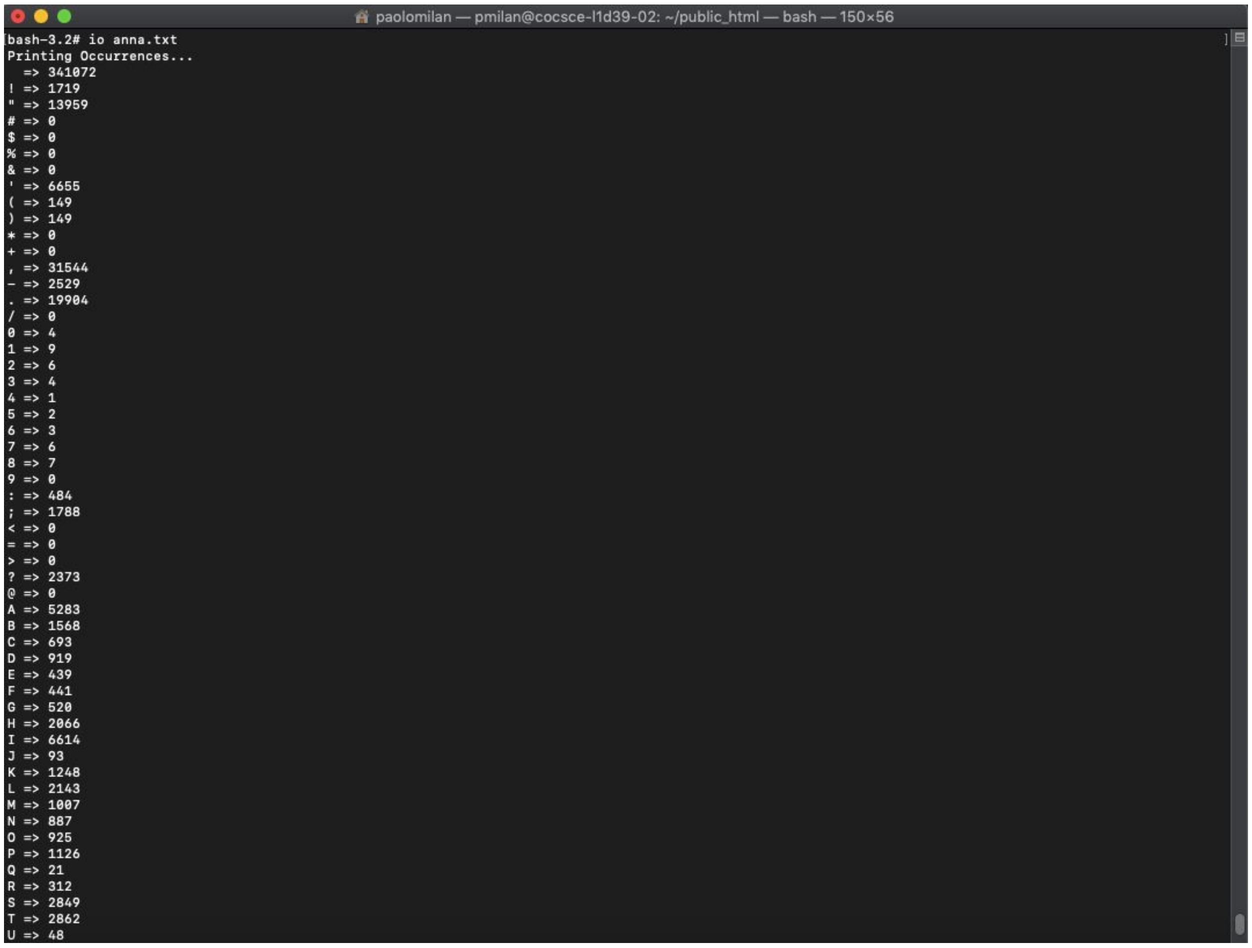
```
int procfile_write(struct file *file, const char *buffer, unsigned long count,
                    void *data)
{
        /* get buffer size */
        procfs_buffer_size = count;
        if (procfs_buffer_size > PROCFS_MAX_SIZE ) {
                procfs_buffer_size = PROCFS_MAX_SIZE;
        }

        /* write data to the buffer */
        if (copy_from_user(procfs_buffer, buffer, procfs_buffer_size) ) {
                return -EFAULT;
        }

        return procfs_buffer_size;
}
```

Patrick Scott
Paolo Milan
CSCE 311 Project 3 Report

2. A loadable kernel module (LKM) is an object file that contains code to extend the running kernel, or so-called base kernel, of an operating system. LKMs are typically used to add support for new hardware (as device drivers) and/or filesystems, or for adding system calls.

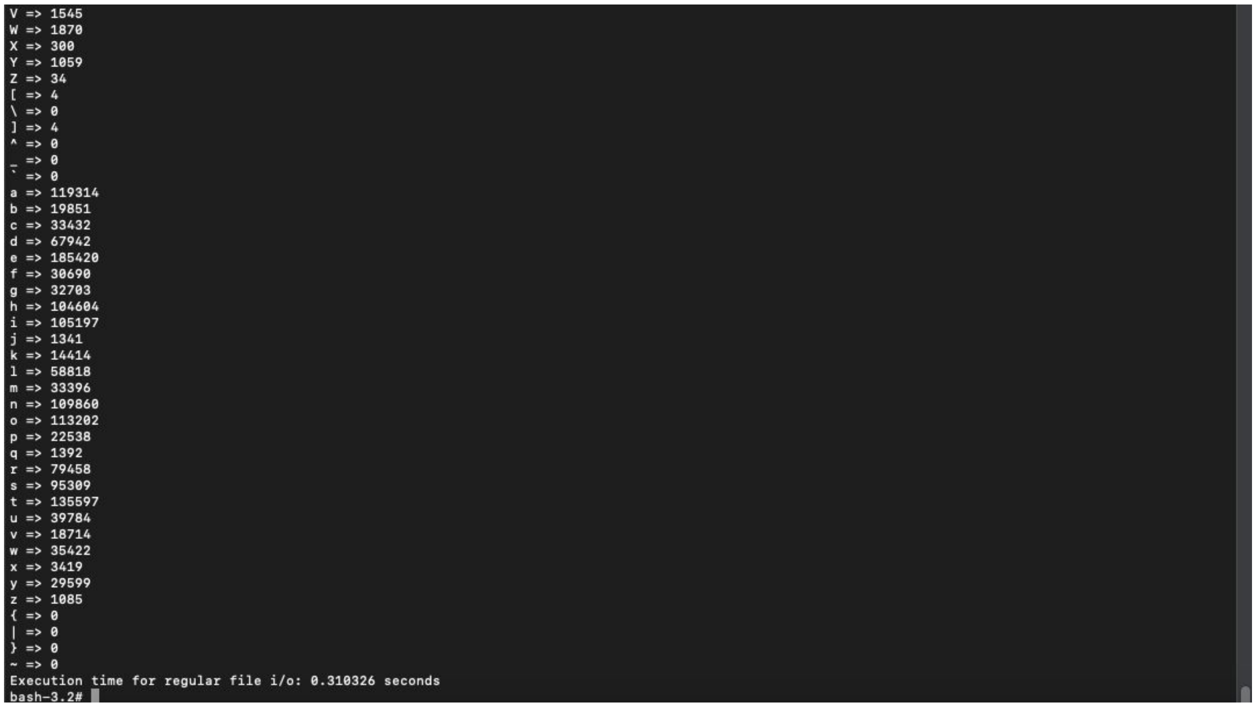
No we cannot use LKM to patch to rebuild a kernel version. It is theoretically possible although it's enormously complex -- every detail would have to be exactly right. Every data structure that increases in size might cause huge areas of memory to need to be reallocated and relocated, and every pointer pointing to one of those pieces of data would then need to be adjusted, and there is a potential cascade of further dependent adjustments.

Sub-Project 1 Results and discussion


paolomilan — pmilan@cocsce-l1d39-02: ~/public_html — bash — 150×56

```
bash-3.2# io anna.txt
Printing Occurrences...
  => 341072
! => 1719
" => 13959
# => 0
$ => 0
% => 0
& => 0
' => 6655
( => 149
) => 149
* => 0
+ => 0
, => 31544
- => 2529
. => 19904
/ => 0
0 => 4
1 => 9
2 => 6
3 => 4
4 => 1
5 => 2
6 => 3
7 => 6
8 => 7
9 => 0
: => 484
; => 1788
< => 0
= => 0
> => 0
? => 2373
@ => 0
A => 5283
B => 1568
C => 693
D => 919
E => 439
F => 441
G => 520
H => 2066
I => 6614
J => 93
K => 1248
L => 2143
M => 1007
N => 887
O => 925
P => 1126
Q => 21
R => 312
S => 2849
T => 2862
U => 48
```

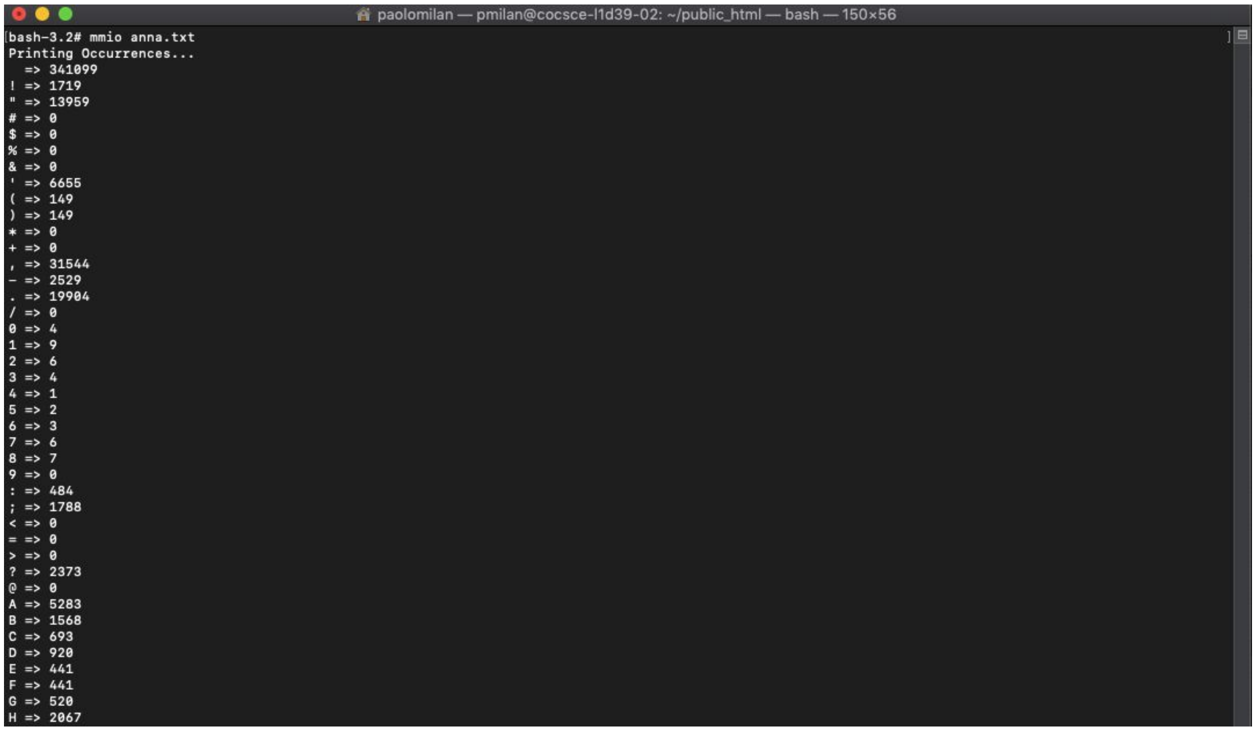
Patrick Scott
Paolo Milan
CSCE 311 Project 3 Report



```
V => 1545
W => 1870
X => 300
Y => 1059
Z => 34
[ => 4
\ => 0
] => 4
^ => 0
_ => 0
` => 0
a => 119314
b => 19851
c => 33432
d => 67942
e => 185420
f => 30690
g => 32703
h => 104604
i => 105197
j => 1341
k => 14414
l => 58818
m => 33396
n => 109860
o => 113202
p => 22538
q => 1392
r => 79458
s => 95309
t => 135597
u => 39784
v => 18714
w => 35422
x => 3419
y => 29599
z => 1085
{ => 0
| => 0
} => 0
~ => 0
Execution time for regular file i/o: 0.310326 seconds
bash-3.2#
```

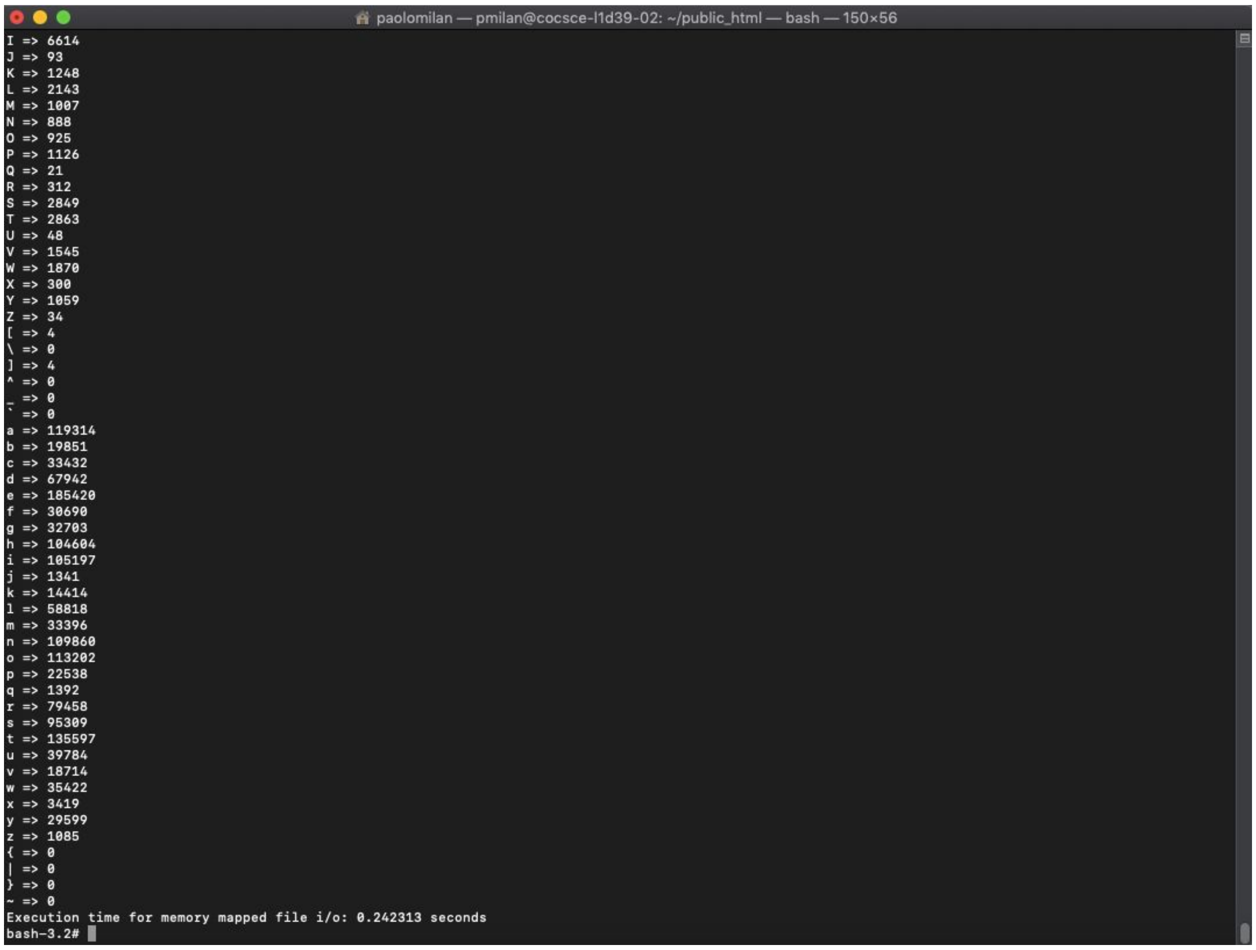
As you can see here, the regular file i.o took 0.310326 seconds.

Next we're gonna look at mmio's execution time.


paolomilan — pmilan@cocsce-l1d39-02: ~/public_html — bash — 150×56

```
[bash-3.2# mmio anna.txt
Printing Occurrences...
  => 341099
! => 1719
" => 13959
# => 0
$ => 0
% => 0
& => 0
' => 6655
( => 149
) => 149
* => 0
+ => 0
, => 31544
- => 2529
. => 19904
/ => 0
0 => 4
1 => 9
2 => 6
3 => 4
4 => 1
5 => 2
6 => 3
7 => 6
8 => 7
9 => 0
: => 484
; => 1788
< => 0
= => 0
> => 0
? => 2373
@ => 0
A => 5283
B => 1568
C => 693
D => 920
E => 441
F => 441
G => 520
H => 2067
```

Patrick Scott
Paolo Milan
CSCE 311 Project 3 Report


paolomilan — pmilan@cocsce-l1d39-02: ~/public_html — bash — 150×56

```
I => 6614
J => 93
K => 1248
L => 2143
M => 1007
N => 888
O => 925
P => 1126
Q => 21
R => 312
S => 2849
T => 2863
U => 48
V => 1545
W => 1870
X => 300
Y => 1059
Z => 34
[ => 4
\ => 0
] => 4
^ => 0
_ => 0
` => 0
a => 119314
b => 19851
c => 33432
d => 67942
e => 185420
f => 30690
g => 32703
h => 104604
i => 105197
j => 1341
k => 14414
l => 58818
m => 33396
n => 109860
o => 113202
p => 22538
q => 1392
r => 79458
s => 95309
t => 135597
u => 39784
v => 18714
w => 35422
x => 3419
y => 29599
z => 1085
{ => 0
| => 0
} => 0
~ => 0
Execution time for memory mapped file i/o: 0.242313 seconds
bash-3.2#
```

As we can see, the execution for a memory mapped file i/o was 0.242313 seconds. On average, this was a lot faster. About 0.07 seconds faster than regular file i/o.

Memory mapped files lets the kernel cache a very large dataset for you using general purpose caching algorithms which make use of all free memory in your system. Additionally, When using memory mapped file you just access the file as it were memory. There is no explicit loading or saving of the file, thus saving time. It isn't much of a difference, but there still is a difference.