Greg Matthews
January 10th 2018

CS 500
Homework #1 - Questions

1. If in the module exit point, we'd forgot to kfree() all of the elements (or part of elements) – what would happen? How do you prevent this from happening? How do you fix this? Show what happens by simulation. Is this different than user level – why or why not? Backup your answer.

Answer:

If one was to forget to kfree() elements that are not being used anymore, then the program will cause a memory leak, meaning a space left in memory still hasn't been freed. If one was to use uninitialized memory and return the contents of that address, it would be random garbage. The more threatening possibility is accidentally overwriting important kernel code when passing around a pointer that forgot to get freed. Overtime with more and more memory leaks due to forgetting to kfree() elements, the main memory of the system (RAM) can become severely handicapped, not being able to use all available memory for other processes, which ultimately degrades performance.

As an example to show this via simulation, I performed kmalloc() of 50,000,000 instances of the person struct from the previous Assignment II code. Before inserting the module, the total size of memory being used by the operating system was 113937 Kilobytes as shown in Fig 1., and after inserting the module, the size of memory being used increased to 1,652,046 Kilobytes as shown in Fig 2. After removing the module without calling kfree() to all the elements, we can see that the size of memory being used is still extremely high as shown in Fig 3, at 1,621,513 Kilobytes because the memory resources have not been freed using kfree(), and now constrains other modules with the amount of memory they can access.

A helpful way to make sure that all memory is successfully freed after program execution, is to keep track of the number of kmalloc() and kfree() calls and make sure they are equivalent. If they are not, then there is a memory leak, or the program is attempting to kfree() an element duplicate times.

```
 Active / Total Objects (% used)    : 527192 / 553995 (95.2%)
 Active / Total Slabs (% used)      : 18553 / 18553 (100.0%)
 Active / Total Caches (% used)     : 76 / 109 (69.7%)
 Active / Total Size (% used)       : 113937.20K / 120368.44K (94.7%)
 Minimum / Average / Maximum Object : 0.01K / 0.22K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
279747 279747 100%    0.10K   7173       39     28692K buffer_head
 48783  36556  74%    0.19K   2323       21      9292K dentry
 37365  35345  94%    1.05K   2491       15     39856K ext4_inode_cache
 28600  28600 100%    0.20K   1430       20      5720K vm_area_struct
 18432  17114  92%    0.03K    144      128       576K kmalloc-32
 18020  18020 100%    0.12K    530       34      2120K kernfs_node_cache
```

Fig 1. List of top kernel resources shown using slabtop, before inserting module

```
 Active / Total Objects (% used)    : 50304325 / 50333879 (99.9%)
 Active / Total Slabs (% used)      : 403527 / 403527 (100.0%)
 Active / Total Caches (% used)     : 76 / 109 (69.7%)
 Active / Total Size (% used)       : 1652046.30K / 1659903.14K (99.5%)
 Minimum / Average / Maximum Object : 0.01K / 0.03K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
50013696 50013696  14%    0.03K 390732      128    1562928K kmalloc-32
 66417  66417 100%    0.10K   1703       39      6812K buffer_head
 48783  35201  72%    0.19K   2323       21      9292K dentry
 37425  35044  93%    1.05K   2495       15     39920K ext4_inode_cache
 28720  28720 100%    0.20K   1436       20      5744K vm_area_struct
 17952  17952 100%    0.12K    528       34      2112K kernfs_node_cache
```

Fig 2. List of top kernel resources shown using slabtop, after inserting module

```
 Active / Total Objects (% used)    : 50274791 / 50295375 (100.0%)
 Active / Total Slabs (% used)      : 401182 / 401182 (100.0%)
 Active / Total Caches (% used)     : 76 / 109 (69.7%)
 Active / Total Size (% used)       : 1621513.34K / 1628384.74K (99.6%)
 Minimum / Average / Maximum Object : 0.01K / 0.03K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
50013952 50013952  14%    0.03K 390734      128    1562936K kmalloc-32
 80301  80301 100%    0.10K   2059       39      8236K buffer_head
 31038  20146  64%    0.19K   1478       21      5912K dentry
 28540  28540 100%    0.20K   1427       20      5708K vm_area_struct
 18054  18054 100%    0.12K    531       34      2124K kernfs_node_cache
 16960  16859  99%    0.06K    265       64      1060K anon_vma_chain
```

Fig 3. List of top kernel resources shown using slabtop, after deleting module

2. Why are the kernel data structures "simple" (for example, linked list, hash, b-trees)? Describe all reasons that you can think of

Answer:

Because the operating system sits in between the application programs such as the compiler and assembler, as well as the hardware, the complexity of code should be as simple as possible to allow for quicker translation of code to hardware components. Similarly to instruction set architecture, a large semantic gap allows for simpler hardware components and faster translation of instruction set to hardware, the operating system's kernel data structures are made simple to allow quicker compilation time from low-level languages to assembly, and finally binary code.

3. What about the security of kernel modules? Can they manipulate global structures or are they limited to their own variables? Suggest what you think the rules are and WHY. How do you get the names of the kernel's variables?

Answer:

Because Kernel Modules hold full kernel privileges, it is of utmost importance to make sure Kernel Modules don't have any posing security risks that could compromise the system. While the

Kernel modules can use global variables and structures, it should be avoided as much as possible to reduce the risk of the global variables changing in other functions and posing security and reliability concerns with the kernel module. Allowing global variables opens up the door for other programs to alter the variable and potentially compromise any of the kernel modules since global variables are shared among all modules of the kernel. One can access the list of kernel variables under the /proc/sys directory, which is managed by specific handlers to allow access for reading and writing. The special system call, called sysctl() gives access to these variables.

4. What about performance (of both the module and the kernel in general)? Since these modules are being loaded into the kernel – are they managed in any way? Can they make the kernel perform badly?

Answer:

Because many modules can be loaded into the kernel, the system manages the loading process using the module-management system. As more modules are being loaded into the kernel, potential memory conflicts can arise, and this issue is managed by the conflict-resolution mechanism. This mechanism allows the system to reserve necessary hardware resources where appropriate to prevent multiple drivers from trying to access the same resource. The conflict-resolution mechanism also manages autoprobes, preventing auto-detect devices from interfering with currently existing devices. The kernel's performance can also degrade overtime with the addition of kernel modules because of the overhead attributed to memory allocation, and resource management complexity.

5. Do you think that a poorly written driver module could cause the kernel to crash (stop working)? Why or why not? How could a kernel stop this from happening?

Answer:

Most certainly, a poorly written driver module can pose a risk of segmentation faults, memory leaks, and dangling pointers can all spell trouble for the kernel and can worst case potentially end up making it crash. This is because kernel modules share the same address space as the kernel itself, meaning each module has the same privileges to overwrite important memory blocks that could crash the kernel, or even corrupt the system. The kernel can gracefully crash however by way of kernel security check failures that start troubleshooting your PC when you run into a problem. Using windows, one such check failure is the blue screen of death. Instead of completely crashing, the kernel will run diagnostics and troubleshoot what may have caused the error as well attempt a recovery of the operating system. A core dump can also be captured by the system, to allow for later analysis of the memory to see what may have gone wrong.

6. Now let's consider the approach of kernels which support loadable modules – what are the tradeoffs (think supportability, performance, security, other areas?) between modular kernels (which allow loadable modules) and monolithic kernels (which don't)? Describe at least three areas of comparison/contrast.

Answer:

Loadable kernels and monolithic kernels have very different use cases and have their own pros and cons depending on the current environment one is working under. If supportability is something that is very important, then loadable kernels will allow a system to be extended in the future by adding support for newer components such as drivers and filesystems. There is however, a downside to loadable kernels that monolithic kernels don't have to worry about, and that is performance degradation due to fragmentation penalties, meaning that whenever new kernel modules are added, the kernel will become fragmented and as a result more calls to cache memory will be needed, and thus the more running time that is necessary. Another advantage of the monolithic kernel is the innate security of kernel modules, since you can't add modules to the system, there is no risk of potentially malicious modules being loaded into your system unsuspectingly, which can potentially be a security risk for loadable modules.