Raul Jimenez

Michael Santella

Project 3

Part 1:

I reused my code to find the task struct using the pid\_task function. If no process is found with the given pid, then the code will print a status message and abort the system call. However, if the process exists then we proceed to get the mm\_struct associated with the virtual memory address which is associated with the process. In this case it is the variable called active\_mm. next we decided to calculate the number of virtual memory areas, so we could check to ensure that every page got printed out through inspection and give us an easy way to iterate through the sorted list given by mmap.

We needed a way to traverse through all of the virtual memory areas, so we decided to the vm\_area\_struct which is used in mmap which is encapsulated within the mm\_struct from active memory. This will give us a sorted list of the virtual memory areas so all we have to do is iterate throught the list.

To find the total size of the virtual memory areas then we must subtract the virtual memory start and end of the virtual memory area. Luckily there is an attribute in the vm\_area\_struct that allows us easily to determine the virtual memory’s area start and end. We will increment the total size variable for each iteration. We would have to subtract the start from the end because that is how address counting works/.

Next, we had to determine permissions. To do this we needed to include the linux/mm.h header file. We learned that if vm\_flags are 1, then it has read permissions, if it has 2 then it has written permissions, and if it has 4 then it has execution permessions. Therefore since, vm\_area\_struct has a vm\_flags variable we could return it and determine what permissions this memory area has. Since it returns an integer then we can determine the access by anding (operator) with 1,2, and 4. Additionally, while traversing though the list we should not increment when we reach the end of the list as it could cause the program to crash. Thus, we need to add a conditional at the end to ensure not to assign the traveler pointer a null pointer.

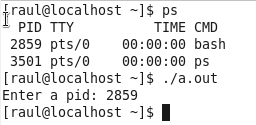
Since each vm\_area\_struct has a vm\_file attribute then we can use this to locate the file the vma belongs to. Since vm\_file is of a type of struct file then we need to include the header file associated with it which is linux/fs.h. it has a struct of type path called f\_path. We can use this with the function under the fs directory called d\_path. We can pass in the vm\_file pointer, as well as a character buffer and the page size. To find the page size we can use the macro defined in linux/page.h. We need to check to see if its null so that we don’t attempt to use d\_path as it could be result in a program crash.

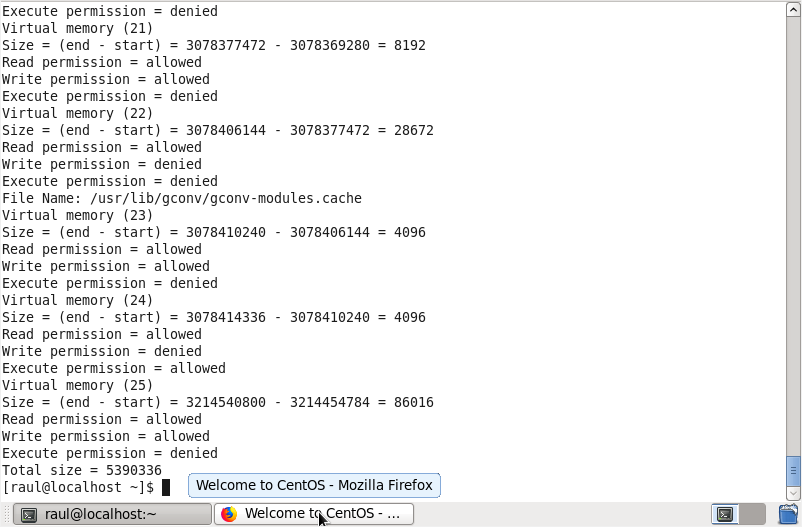
The second argument of the d\_path function requires a character pointer, and according to the documentation. Since I just want a character buffer for the file, I will choose something small and just get the next page to temporarily store the file name. all d\_path does is convert a dentry into an ASCII name path.( <https://www.kernel.org/doc/htmldocs/filesystems/API-d-path.html>).

Next, we followed the instructions from project 1 to add the system call to the kernel and reused code from project 1 to make the test case. Something that we noticed after running the test code on bash is that not every virtual memory area has a file mapped to it.

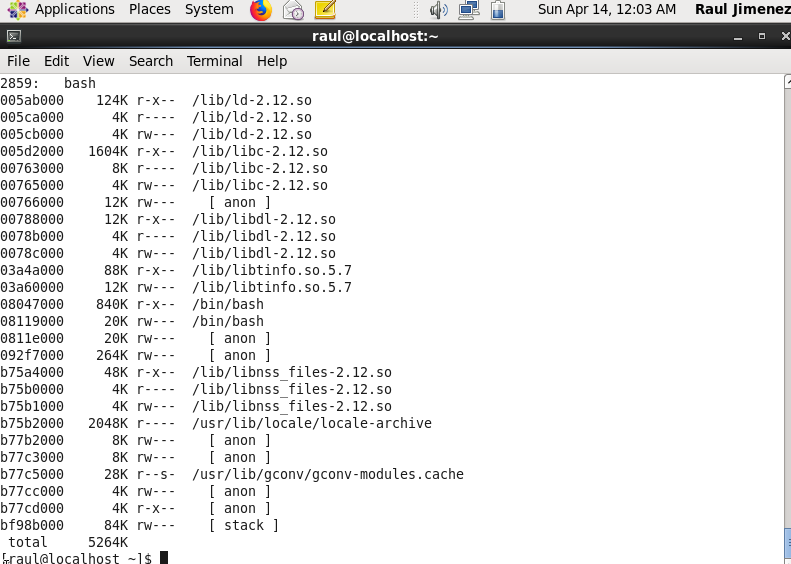
To compile the code please do gcc *finalname.c*

Results:





Verification:



Part 2:

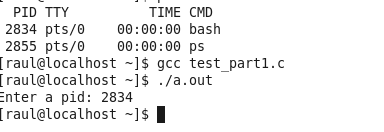
Since we need to report on the status of a specific address, then we need two arguments, the virtual memory area and the process id. I found the task\_struct the same way I did in part 1, using pid\_task(). Since we are access the vma then we need to have a pointer to the attribute active\_mm.

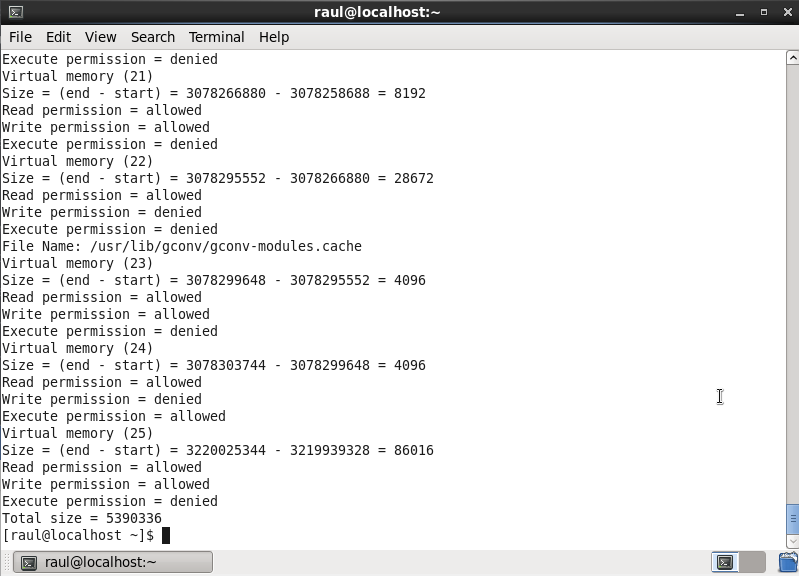
After reading the page table handling section from the book ULK (3rd edition), I found out the function that I can use to check each individual page for reference, dirty, and active bits. However, to get the page offset we must walk the page table so that we can get the offset. I am using the documentation from (<https://www.kernel.org/doc/gorman/html/understand/understand006.html>) as for reference my code. I was getting the incorrect offset using the documentation code, so I found the solution to the problem in stack overflow. (<https://stackoverflow.com/questions/41090469/linux-kernel-how-to-get-physical-address-memory-management>) I was not referencing the upper directory when offsetting the middle directory. I then called the proper function to get the upper directory offset in order to properly walk through the memory.

To use the functions given in the documentation I need to include the header file asm/pgtable.h. Additionally to use the page variables we need to include the header file asm/page.h. After adding the system call similarly to part 1, I recompiled the kernel and reinstalled it to test it with my test program.

To determine if the memory was in memory or not we used the function pte\_present() because it is already built into the kernel. If it returns 1 then it is in memory otherwise if it returns 0 then it is not in memory and is thus on disk. To determine if the memory area was referenced then we would use the function pte\_young(). It would return 1 if the memory area was accessed recently, and 0 if the memory was not referenced. Lastly there is function pte\_dirty() which will return 1 if the dirty bit is on and 0 if the dirty bit is off. Thus, since these are true or false return values, it is easy to use conditionals to print out the information. As always to compile the code use gcc filename.c.

Results:





And using virtual memory address 3220025344.

