Resource containers -- memory

Overview

In the previous project, you have experienced the power of adding a new abstraction, resource containers, in the system and use that as another way to schedule/allocate computing resource for tasks (e.g., processes, threads). In addition to processors, memory is another resource that tasks can share and resource containers can potentially help to manage.

In this project, we are going to extend the concept of resource containers to memory. A resource container can hold a few physical memory locations. Tasks within the same resource container can share these pool of memory locations upon requests. On the other hand, tasks do not belong to the same container cannot share memory locations, even though they are spawned from the same process.

To implement this abstraction, we plan to use a kernel module, as what you have done in the previous project. This kernel module supports a few ioctl commands that allow user-space library/applications to initialize a device, allocate memory locations, or assign a task to a resource container. The library interacts with both the application and the kernel module to translate the requests and responses between them.

You will be given the prototype of the kernel module with a core.c and ioctl.c file in its source directory that only contains empty functions. We also provide a user-space library that allows an application to interact with this kernel module through ioctl interfaces as well as a sample benchmark application that you may extend to test if your kernel module functions correctly.

You are strongly encouraged to work in a group of 2. Groups do the same project as individuals. Both members will receive the same grade. Note that working in groups may or may not make the project easier, depending on how the group interactions work out. If collaboration issues arise, contact your instructor as soon as possible: flexibility in dealing with such issues decreases as the deadline approaches.

Objective

* Learning UNIX/Linux kernel programming as well as the constraints
* Learning UNIX/Linux system memory management
* Learning UNIX/Linux kernel modules
* Learning multithreading/multiprogrammed

How to start

To begin, you need to first form a group and setup the environment for developing your project. You should set up a machine or a VMWare virtual machine (CS students should have the free license for that https://www.csc.ncsu.edu/vmap/) with a clean Ubuntu 16.04 installation. You also may use the VCL service maintained by NCSU through https://vcl.ncsu.edu/. You may reserve one virtual machine and connect to this machine remotely by selecting reservations. We will use the "Ubuntu 16.04 Base" to test your kernel module. However, the VCL service will reset once your reservation timeout.

Then, you need to clone the code from https://github.ncsu.edu/htseng3/CSC501_Container_Memory and make your own private repository. Please do not fork for the given repository. Otherwise, you will be the public repository. After cloning the code, you will find three directories and a test script. These directories are:

1. kernel_module -- the directory where we have the kernel module code.

2. library -- the directory of the user-space library code.

3. benchmark -- the directory with a sample program using this kernel.

You may now go to the kernel_module directory and type "make" to compile the kernel module and then "sudo make install" to install headers and the module in the right place. You should be able to find a memory_container.ko if your compilation success and this is the binary of the processor_container kernel module.

However, this kernel module isn't in your kernel yet. To get this kernel module loaded into the system kernel, try "sudo insmod memory_container.ko". Upon success, you should find an "mcontainer" device file under /dev directory in your linux system. By default, this device may not be available for non-root users. Therefore, you need to use "sudo chmod 777 /dev/mcontainer" command to make it accessible by anyone and any process in the system.

If you don't want this device to be available in the system anymore, you can use "sudo rmmod memory_container" to remove this device.

Now, you can navigate to the library path and again use "make" to generate this dynamic link library. You need to then use "sudo make install" to make this library publicly available for the system. You should read the code and figure out how this library interacts with the kernel module.

Finally, you can now go to the benchmark directory to get the benchmark program compiled and use the shell script "test.sh" at the base folder to test and validate your implementation.

For example, if type "./test.sh 128 4096 1 1", it will generate 1 container which is registered by 1 task, and it will have 4096 bytes data in each object and 128 objects in total. If type "./test.sh 128 4096 2 1", it will generate 1 container which is registered by 2 tasks, and it will have 4096 bytes data in each

object and 128 objects in total and those objects are shared by these tasks in the same container. So on and so forth.

No matter you're using VMWare, a real machine, or VCL, you should always use https://github.ncsu.edu to control/maintain/backup your work.

Your tasks

1. Implementing the memory_container kernel module: it needs the following features:

- **create**: you will need to support the create operation that creates a container if the corresponding cid hasn't been assigned yet, and assign the task to the container. These create requests are invoked by the user-space library using **ioctl** interface. The ioctl system call will be redirected to memory_container_ioctl function located in src/ioctl.c

- **delete**: you will need to support delete operation that removes tasks from the container. These delete requests are invoked by the user-space library using **ioctl** interface. The ioctl system call will be redirected to memory_container_ioctl function located in src/ioctl.c

- **mmap**: you will need to support **mmap**, the interface that user-space library uses to request the mapping of kernel space memory into the user-space memory. The kernel module takes an offset from the user-space library and allocates the requested size associated with that offset. You may consider that offset as an **object id**. If an object associated with an offset was already created/requested since the kernel module is loaded, the mmap request should **assign** the address of the previously allocated object to the mmap request. The kernel module interface will call memory_container_mmap() in src/core.c to request an mmap operation. One of the parameters for the memory_container_mmap() is "struct vm_area_struct *vma". This data structure contains page offset, starting virtual address, and etc, those you will need to allocate memory space.

- **lock/unlock**: you will need to support locking and unlocking that guarantees only one process can access an object/a set of memory pages at the same time. These lock/unlock functions are invoked by the user-space library using **ioctl** interface. The ioctl system call will be redirected to memory_container_ioctl function located in src/ioctl.

- **free**: you will need to support delete operation that removes an object from memory_container. These delete requests are invoked by the user-space library using **ioctl** interface. The ioctl system call will be redirected to memory_container_ioctl function located in src/ioctl.c

2. Test the developed module: It's your responsibility to test the developed kernel module thoroughly. Our benchmark is just a starting point of your testing. The TA/grader will generate a different test sequence to test your program when grading. Your module should support an infinite number of containers and different numbers of tasks with each container.

Turn ins

You **only need to (or say you can only) turn in the core.c and the ioctl.c** file in the kernel_module/src directory as a tarball. All your modifications should be limited within these two files. Exactly 1 member of each group should submit the source code. All group members' names and Unity IDs should be easily found in a single line comment at the beginning of the code in the following format:

// Project 2: 1st member's name, 1st member's Unity ID; 2nd member's name, 2nd member's Unity ID

You need to name the tarball as {1st_members_unityid}_{2nd_members_unityid}_mcontainer.tar.gz

Reference and hints

1. You should try to figure out the interaction between user-space applications (e.g. benchmark, validate) and the library, the library and the kernel module. You should especially understand how mmap is used in mcontainer_alloc function from the library. Here is the explanation of mmap function http://man7.org/linux/man-pages/man2/mmap.2.html

2. You may need to reference the Linux kernel programming guide and Linux Device Drivers, 3rd Edition since user-space libraries will not be available for kernel code.

3. You may find the following kernel library functions useful

- kmalloc/kfree

- remap_pfn_range

- copy_from_user

- mutex_lock/mutex_unlock

- virt_to_phys

Submission status

Attempt number	This is attempt 1.
Submission status	No attempt
Grading status	Graded
Due date	Friday, November 2, 2018, 12:00 AM
Time remaining	Assignment is overdue by: 126 days 14 hours
Last modified	-

Submission comments

Comments (0)

Add submission

Make changes to your submission

Feedback

Grade	100.00 / 100.00
Graded on	Sunday, November 4, 2018, 10:40 PM
Graded by	Qi Zhao

◄ Resource containers -- processor

Jump to...

Resource containers -- file ►

Data retention summary
Get the mobile app