

CSC 501 (001) Fall 2018 Operating Systems Principles

WolfWare / Dashboard / My courses / CSC 501 (001) FALL 2018 / Projects
/ Resource containers – processor

Resource containers – processor

Overview

Modern operating systems promote virtualization of their underlying machines and isolation using process/thread abstractions. As a result, threads/processes also become the identities of resource scheduling for tasks in the system. However, the "matching" between abstraction and resource allocation also creates "mismatching" between tasks and their real demands of resources. For example, process scheduling is based on the CPU time, regardless their I/O usages. The system also lacks flexibilities in controlling the usage of system resources for threads/processes with different properties as they are all treated the same.

To address the above issue, **resource containers** provide another abstraction other than processes and threads for resource allocation. Each resource container logically abstracts a set of system resources for tasks within the container to use. Depending on the demand of applications, the system can assign different resource containers with different amount of system resources. Each resource container can potentially implement its own scheduling policy to efficiently use its own resources. Recent cloud platforms as well as software engineering platforms further extend the concept of containers to achieve lightweight virtualization and protection among tasks.

As operating systems only supports processes and threads by default, implementing resource containers would require additional efforts in any operating system kernel. Fortunately, most modern operating systems support "loadable kernel modules". In this way, the system can boot with a simpler, smaller kernel and then load these modules into kernel space when necessary. In this project, we will implement resource containers as a loadable kernel module as well as set of library functions that create a pseudo device in the system and provide an interface for applications. By interacting with this device, processes can assign its own threads to difference resource containers.

With this new facilities, threads assigned to the same resource container will/can only share the same set of resources within the container. This semester, you will be gradually building this new facilities in terms of the supports for resource allocations for processors, memory and file storage. In the first project, you will build a kernel module to allocate processor resources and schedule the execution of threads within resource containers.

In this project, you will be given the prototype of the kernel module with a `core.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 process scheduling
- * Learning UNIX/Linux kernel modules
- * Learning multithreading programming
- * Learning UNIX/Linux interrupt handler implementation

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 free license for that <https://www.csc.ncsu.edu/vmap/>) with 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_Processor 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 `processor_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 `processor_container.ko`". Upon success, you should find an "pcontainer" 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/pcontainer`" 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 processor_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. The benchmark take input parameters for generating the corresponding number of containers and the number of task in each container.

For example, if type "./benchmark 1 1", it will generate 1 container, and the first container will have 1 task inside. If type "./benchmark 2 2 4", it will generate 2 containers, and the first container will have 2 tasks and the second container will have 4 tasks. So on and so forth.

The benchmark program creates containers and tasks, runs some simple calculations and prints out how many time the task runs the calculations. The number of runs among tasks inside the same container should be similar (within 10% error), and the total runs among containers should be also similar (within 10% error). You may go into the benchmark.c to understand the meaning of the program. You should also use and read "test.sh" file to see how the whole framework work together.

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 process_container kernel module: it needs the following features:

- **create**: you will need to support 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 process_container_ioctl function located in src/ioctl.c
- **delete**: you will need to support delete operation that removes tasks from the container. If there is no task in the container, the container should be destroyed as well. These delete requests are invoked by the user-space library using **ioctl** interface. The ioctl system call will be redirected to process_container_ioctl function located in src/ioctl.c
- **switch**: you will need to support Linux process scheduling mechanism to switch tasks between threads.
- **lock/unlock**: you will need to support locking and unlocking that guarantees only one process can access an object 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 process_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 1: 1st member's name, 1st member's Unity ID; 2nd member's name, 2nd member's Unity ID; 3rd member's name, 3rd member's Unity ID;
```

You need to name the tarball as

```
{1st_members_unityid}_{2nd_members_unityid}_{3rd_members_unityid}_npheap.tar.gz
```

Reference and hints

1. This project is based on this

paper: http://people.cs.uchicago.edu/~shanlu/teaching/33100_fa15/papers/rc-osdi99.pdf, you may need to read it to understand the high-level view of this project.

2. You may also search Linux cgroup, LXC or Docker implementations to get deeper understanding of this project.

3. You should try to figure out the interactions between user-space applications (e.g. benchmark) and the user-space library, the user-space library and the kernel module. You should especially understand how to context switch tasks in the user-space that the functionality is defined in pcontainer_init(), handler(), pcontainer_context_switch_handler() from the user-space library. And you also need to know how to wake up/pause tasks by using wake_up_process()/schedule()/set_current_state(), which are kernel-space functions. Here is the explanation of how to control the status of processes in Linux system: <https://www.linuxjournal.com/article/8144>

4. 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.

5. You may find the following kernel library functions useful

- kmalloc/kcalloc/kfree
- mutex_init/mutex_lock/mutex_unlock
- wake_up_process/schedule/set_current_state
- printk
- copy_from_user

6. You may need to know these variable


```
volatile long TASK_INTERRUPTIBLE;
```


```
volatile long TASK_RUNNING;
```

```
struct task_struct *current;

gfp_t GFP_KERNEL;
```

Submission status


Submission status	Submitted for grading
Grading status	Graded
Due date	Friday, September 28, 2018, 12:00 AM
Time remaining	Assignment was submitted 10 hours 16 mins early
Last modified	Thursday, September 27, 2018, 1:43 PM
File submissions	<div><div> ssharm34_bkarumu_npheap.tar.gz +</div><div>Export to portfolio</div></div>

Submission comments	<div><div> Comments (0)</div></div>
---------------------	--

Edit submission

Make changes to your submission

Feedback

Grade	100.00 / 100.00
Graded on	Sunday, October 7, 2018, 11:30 PM
Graded by	<div><div></div><div>Qi Zhao</div></div>

[◀ Announcements](#)[Resource containers -- memory ▶](#)

You are logged in as Shantanu Sharma (Log out)

CSC 501 (001) FALL 2018

[English - United States \(en_us\)](#)

[Deutsch \(de\)](#)

[English - United States \(en_us\)](#)

[English \(en\)](#)

[Español - Internacional \(es\)](#)

[Français \(fr\)](#)

[Português - Brasil \(pt_br\)](#)

[اردو \(ur\)](#)

[हिंदी \(hi\)](#)

[简体中文 \(zh_cn\)](#)

[Data retention summary](#)

[Get the mobile app](#)