COL331/COL633: Operating Systems
Assignment 3

Total: 20 marks
Due Date: 22 April 2019, 23:59

Key Points:

1. **The assignment will be in groups of 2 students.**
2. -15% penalty per day, after the deadline.
3. We shall use Ubuntu Linux 18.04 to test the assignment. If the assignment does not work, it is the student's fault.

1 Virtualization; VM vs Container: A quick tutorial

> "In computing, virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources." – Wikipedia

Virtualization is used to provide isolation, flexibility, higher resource utilization, easy resource management as well as resource elasticity and to enable co-existence of heterogeneous services on the same physical hardware.

The traditional way of running a VM was to install a hypervisor on a host machine and then install multiple OS on top of that. The resources are managed by the hypervisor, as shown in Figure 1 (a). Recent trends have shifted from a hypervisor based virtualization to a container based virtualization. In a container based virtualization, the OS acts as a base and supports various containers running on top of it. These containers are used to provide virtualization for applications. Please note that in this case the OS remains the same across all the containers and the resource management between the containers is managed by the base OS. The applications running in different containers, see an isolated view of the OS as shown in Figure 1 (b).

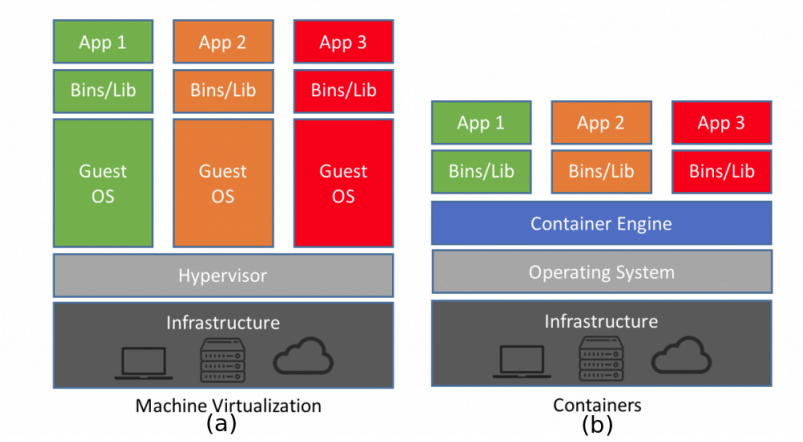

Figure 1: Different between a virtual machine and a container.

1.1 cgroups and LXC

Control groups, popularly known as *cgroups*, is a kernel feature that provides the resource isolation of CPU, memory etc. among different process groups. Please find more details on the Wikipedia page [1]. LXC or Linux containers is an implementation of containers within the Linux kernel. This was the first effort at providing a light weight version of virtualization. LXC uses cgroups and namespace isloation property of the Linux kernel to provide support for containers. Please find more details on the Wikipedia page [2].

2 Virtualization in XV6

XV6 does not support virtualization, as this is a toy OS with basic constructs available. The purpose of this assignment is to familiarize with the concepts of virtualization by implementing it in XV6. These implementations will mimic the functionalities provided by a *container* and will NOT be a full fledged *container* service.

Please note that in an actual container implementation, the kernel is not aware of the processes created and running inside a container and it is the responsibility of the *container* to handle all the OS related operations, for the processes running inside that container, such as the memory manager, file system, system call handling etc. However, implementing all the features in XV6 is out of scope for this course.

In this assignment you will implement container related services to support the following features:

- Container manager.
- Virtual scheduler.
- System calls.
- Resource isolation:
  - ~~Virtual page table~~, and
  - Virtual file system.

These are explained in detail in the subsequent sections.

2.1 Container manager

Container manager is a user program which will mimic the functionalities provided by a container service. Name this user program as **container_manager.c**. The basic functionalities that should be provided are:

- Maintenance of the data structures required for the container service.
- Scheduling of the processes within the container.

2.1.1 Container creation

You need to implement a system call which will create a container. Container will be a user program, container manager, as explained in the Section 2.1. Please note that there can be more than one container running at the same time, and each of them should be isolated from each other. This means that data structures maintained by the container manager should be local to a specific container.

You need to implement system call which will register the container with the kernel, *create_container* and *destroy_container*.

```
uint create_container(void)
uint destroy_container(uint container_id)
```

The system call **create_container**, will create a container and returns the container_id of the container if the operation is successful. It returns a negative number if the operations fails. Please note that the container_id is unique across the containers.

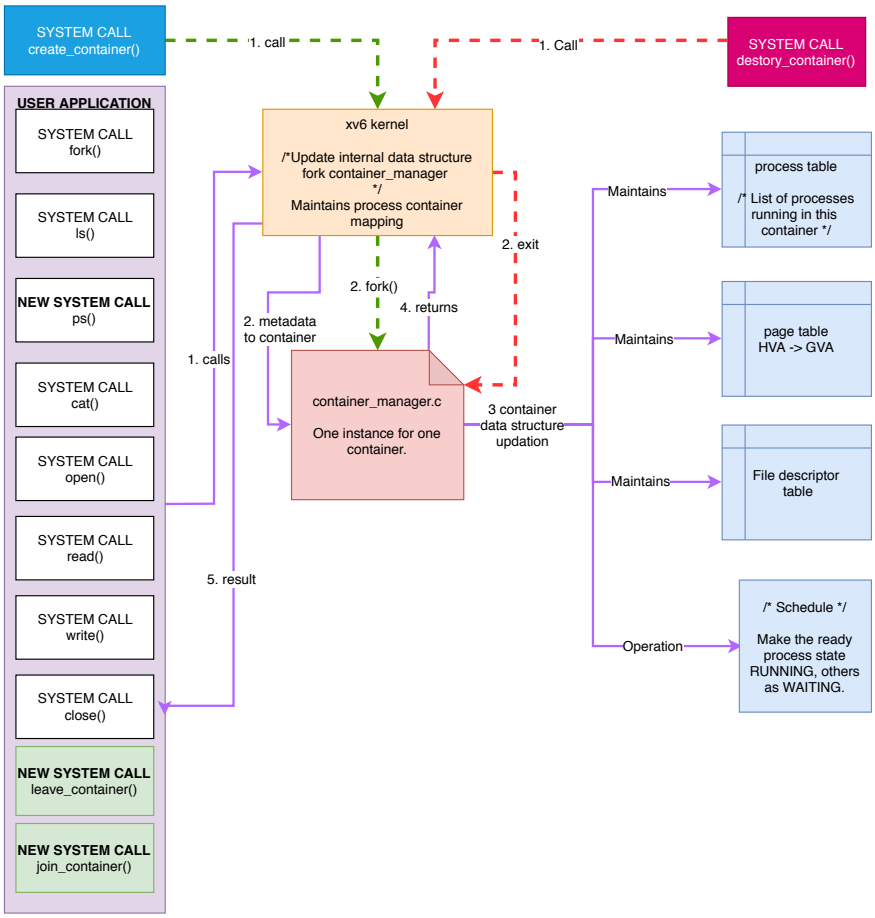

Figure 2: High level design of the virtualization in XV6. Color coded arrows represent an operation, with number showing the order.

2.1.2 Joining a container

The processes will be created using the old *fork* command. You need to implement two new system calls, *join_container* and *leave_container*. As the name suggests, these will be used by processes to join and leave a container respectively. Please note that at a given time a process can only belong to a single container. The format of the system call will be:

```
uint join_container(uint container_id)
uint leave_container(void)
```

Please note that the XV6 kernel will be aware of this process as this virtual system call will internally call the *fork* system call, however, the scheduling will be done by the container service (by manipulating the readiness of the process or any other method you deem fit for the purpose).

2.2 Virtual Scheduler

In a container, there can be more than one process can be running at a given time. It is the responsibility of the *container* to schedule them.

As we are mimicking a *container* and not a complete system, the process created within the container is visible to the xv6 kernel and xv6's scheduler will try to schedule it. To prevent this we will keep the state of the process as WAITING and only change it to READY when the container scheduler decides to run this process. Hence, the *container* need to maintain the STATE (called V_STATE) of the processes within its local data structure and not rely on the STATE present inside the PCB (process control block).

```
1 // Per-process state
```

```
2  struct proc {
3  uint sz;                     // Size of process memory (bytes)
4  pde_t* pgdir;                // Page table
5  char *kstack;                // Bottom of kernel stack for this process
6  enum procstate state;        // Process state <-------------------------------
7  int pid;                     // Process ID
8  struct proc *parent;         // Parent process
9  struct trapframe *tf;        // Trap frame for current syscall
10 struct context *context;     // swtch() here to run process
11 void *chan;                  // If non-zero, sleeping on chan
12 int killed;                  // If non-zero, have been killed
13 struct file *ofile[NOFILE];  // Open files
14 struct inode *cwd;           // Current directory
15 char name[16];               // Process name (debugging)
16 };
```

Listing 1: XV6 PCB, see file proc.h. See line 6 for process state.

You might need to modify the PCB state to include the container information. The scheduling policy will be a simple round robin, similar to the XV6 scheduler. You are free to modify this, please mention this in your report and the reason for choosing that particular policy over the round robin algorithm.

2.3 System calls

In an actual container implementation, the system calls are trapped by the container and are issued on behalf of the process after verification (whether the process is allowed to make this system call or not). However, in our case if the process makes a system call, it will directly go to the XV6 kernel. You will need to make appropriate changes to the kernel so that the *container* of that particular process is aware of the process activities and the container manager has the capability of blocking certain system calls if the process making that call is not authorized to make it.

It is the responsibility of the kernel to provide the appropriate isolation between two containers. That is processes running in one container should not be visible in another container. Also, once a process has joined a particular container it should not be visible to the host machine and any changes made by that process henceforth also should be hidden from the host machine.

2.4 Resource Isolation

Containers are isolated from each other by using the mechanisms provided by the kernel. Please note that, for file systems, the containers have a common view of the host file system at the time of their creation. However, any changes made to the file system by a container is local to that container.

2.4.1 Page Table

~~Processes running in a virtual environment, generate virtual addresses corresponding to the container (guest virtual address or GVA). These addresses are resolved to container physical address via the page table present inside the container. However, this is not enough. The addresses need to be resolved to the host physical addresses and hence there is one more page walk in the host system's page table.~~

~~To simulate this, the container manager will have a *page_table* structure, which will be filled upon using the *container_malloc* virtual system call. The value returned by the *container_malloc* call will be a GVA (guest virtual address). This GVA will be used by the processes for their activities. To resolve a GVA it has to go following transitions.~~

~~GVA → GPA → HVA → HPA.~~

~~Here:~~
~~GVA: Guest virtual address~~
~~GPA: Guest physical address~~
~~HVA: Host virtual address~~

~~HPA: Host physical address~~
~~**Note:** For most cases GPA = HVA.~~
~~A general guest page table structure can look like as shown in Table 1.~~

~~GVA~~	~~GPA/HVA~~	~~pid~~
~~a~~	~~0xfabd~~	~~1~~
~~b~~	~~0x1450~~	~~2~~

Table 1: ~~A general design for a global page table inside the container. Please note that you can change it in anyway you want for you implementation. The content of the page table can also be changed.~~

2.4.2 Virtual File system

As with the memory management, a container provides support for file system also. The system calls are trapped and served by the container.

In this case also, we will modify the system calls (open, read, write and close) which will be used by the processes running inside the container for their operations. The returned value of these calls will be a **cfd** ,i.e. a container file descriptor. This will a one to one mapping with fd returned by the XV6 kernel. A table, similar to the page table, is used to maintain this key value pair relationship as shown in Table 2.

cfd	fd	pid
1	2	1
3	1	2

Table 2: A general design for a global page table inside the container. Please note that you can change it in anyway you want for you implementation. The content of the page table can also be changed.

3 Evaluation criteria

The basic construct of a container is that it provides isolation to the applications running inside a container. Isolation with respect to other processes running on the system and file system content. The evaluation criteria is:

Standard operations [4 Marks]: Please note that all the three operations should work in this case. There will be no partial marks.

1. Create a container.
   The system call *create_container(uint container_id)* should return without any error.
2. Join a container.
   The system call *join_container(uint container_id)* should return without any error.
3. Leave a container.
   The system call *leave_container(void)* should return without any error.
4. Destroy a container.
   The system call *destroy_container(uint container_id)* should return without any error.

Command output [12 Marks]:

1. Output of the *ls* command [**4 marks**]:
   This command, which is already present in XV6, prints the name of the files in the current directory. You need to modify the system call so that it respects the isolation provided by the containers. The files created by processes on container A should not appear in the output of the *ls* command issued from any other container.
2. Output of the *ps* command [**4 marks**]:
   You need to implements this system call. This prints all the current RUNNING or WAITING process from the process queue. The process started (joined) within a container should be visible to processes inside that container only and not to other processes running in some other container.

3. Copy-on-write or COW mechanism [**4 marks**]:
   Two processes in two different containers can modify the same files, which was present in the host file system and hence visible to both the processes in different containers. However, as soon as the processes start to modify the file, the changes made to the file should be local to the container and should not be visible outside the container.

3.1 Test Script

You also need to write a user program of the following format which will test all the required features. Name the file as **user_test_assig3.c** .

```
/* Creating the containers. */
create_container(1)
create_container(2)
create_container(3)

/* Three container managers (user programs) should be running now */

// ============================================================
/* Multiple process creation to test the scheduler */
fork()
join_container(1) \\ called only by child created by preceeding fork call.
fork()
join_container(1) \\ called only by child created by preceeding fork call.
fork()
join_container(1) \\ called only by child created by preceeding fork call.

/* Testing of resource isolation */
fork()
join_container(2) \\ called only by child created by preceeding fork call.
fork()
join_container(3) \\ called only by child created by preceeding fork call.


/*---------- PROCESS ISOLATION ------------------*/
// called by atmost one process in every container.
ps();

/* ---------------SCHEDULER TEST------------------------ */
scheduler_log_on(); // This will enable logs from the container scheduler
/* Print statements of form (without the quotes): */
"Container + <container_id> : Scheduling process + <pid>"
scheduler_log_off(); // Disable the scheduler log after scheduling all the child process
    atleast once.

/* ----------------FILE SYSTEM TEST----------------------- */
/* Executed by all the child processes in all the containers */
ls() // This will print the host file system
create("file_"+pid) // pid can be same across containers, however, in this case they
   will be different as they are created using fork and join a container later.

// BARRIER // All the process should finish the create system call

ls() // the container should see files created by processes running inside in it along
   with the original files from the host system.

/* Executed by only one child process in every container*/
open("my_file");
write("Modified by: "+pid);
close("my_file");
cat("my_file"); // The contents of the file should be different for process running
```

```
    across containers.

/* Executed by all the child processes */
leave_container()

// ============================================================

// Executed by the main process
destroy_container(1)
destroy_container(2)
destroy_container(3)

exit();
```

Listing 2: Pseduocode to test the complete system

4 Report [4 marks]

Please include a report PDF format detailing the logic behind the operations mentioned in Section 3. Please limit the total number of pages to 5.

5 Submission Instructions

- We will run MOSS on the submissions. We will also include last year's submissions. Any cheating will result in a zero in the assignment, a penalty as per the course policy and possibly much stricter penalties (including a fail grade and/or a DISCO).
- There WILL be a demo for assignment 3.

How to submit:

1. Copy your report in the xv6 root directory.
2. Then, in the root directory run

   ```
   make clean
   tar czvf assignment3_<entry_number1>_<entry_number2>.tar.gz *
   ```

   This will create a tar ball with name, assignment3_⟨entry_number1⟩_⟨entry_number2⟩.tar.gz in the same directory. Submit this tar ball on moodle.
3. **ONLY ONE SUBMISSION PER GROUP.**

References

[1] cgroups - wikipedia. `https://en.wikipedia.org/wiki/Cgroups`. (Accessed on 03/23/2019).

[2] Lxc - wikipedia. `https://en.wikipedia.org/wiki/LXC`. (Accessed on 03/23/2019).