COL331 Assignment-3 Report

Pranav Bhagat - 2016CS10352
Sachin Kumar Prajapati - 2016CS10355

April 26, 2019

Data Structure for Container

- The following struct is added to *proc. h* to give us the functionality of containers:-

```
38  struct container{
39    uint id;
40    int curproc;
41    int pids[NPROC];
42    char file_names[10][10];
43    int file_count;
44  };
45
```

- Following are the descriptions of the variables in the struct:-
  - **uint id**: The unique id of containers starting from 1.
  - **int curproc**: The pointer inside the container which tells about the last process which was scheduled in the container. Helps in maintaining Round Robin scheduling scheme inside the containers.
  - **int pids[NPROC]**: This list of the pids of the process inside the container, where **NPROC=64** is the maximum no. of process inside a container.
  - **char file_names[10][10]**: A list of char arrays which stores the names of files made inside the container. This is used when we want to destroy a container to delete the files present inside that container.
  - **int file_count**: Stores the number of files made inside the container.
- In addition to this, a variable ***container_id*** denoting the container id of processes is added(Line 64) to the struct of processes in *proc.h*.

```
48  // Per-process state
49  struct proc {
50    uint sz;                     // Size of process memory (bytes)
51    pde_t* pgdir;                // Page table
52    char *kstack;                // Bottom of kernel stack for this process
53    enum procstate state;        // Process state
54    int pid;                     // Process ID
55    struct proc *parent;         // Parent process
56    struct trapframe *tf;        // Trap frame for current syscall
57    struct context *context;     // swtch() here to run process
58    void *chan;                  // If non-zero, sleeping on chan
59    int killed;                  // If non-zero, have been killed
60    struct file *ofile[NOFILE];  // Open files
61    struct inode *cwd;           // Current directory
62    char name[16];               // Process name (debugging)
63    //**********************
64    uint container_id;           // Container Id
65    //**********************
66  };
```

Standard Operations

Containers are implemented in kernel space.

a) Creating a container

- A system call *create_container(void)* is added for this operation which returns 0 on success and -1 otherwise.
- On invocation of this call, a new struct for container is declared and is added to the global list of containers in the kernel.

b) Join container

- A system call *join_container(uint container_id)* is added for this operation which returns 0 on success and -1 otherwise.
- On invocation of this call, first we check if the container could accommodate this process or not, if it can the container_id of the process is set to the given id and the pid of the process is added to the list of pids in container.

c) Leave container

- A system call *leave_container(uint container_id)* is added for this operation which returns 0 on success and -1 otherwise.
- On invocation of this call, the container_id of the process is reset and the pid of the process is removed from the list of pids in the container.

c) Destroying container

- A system call *destory_container(uint container_id)* is added for this operation which returns 0 on success and -1 otherwise.

- On invocation of this call, first we access the list of pids of the container and kill all the processes inside this container. After that we remove the container from the global list of containers in the kernel. Also, the files made by a process in the container are deleted on destroying the container.

Scheduling of processes

- Scheduling of process is done in **Round Robin Manner**. For implementing this we have tweaked the working of XV6 scheduler.
- In original scheduling, the kernel loops through all the process to find a process which can be scheduled and then schedules it. But now during this search for schedulable process, if the process is part of any container we just leave it and continue with our search.
- Now after the completion of the regular for loop which scheduler uses. We have added one more for loop which iterates over all containers and asks them which process they want to schedule. Now it is the scheduler's responsibility to maintain Round Robin scheduling among its processes.
- The container maintains Round Robin Scheduling among its process by maintaining a pointer to last process it scheduled.

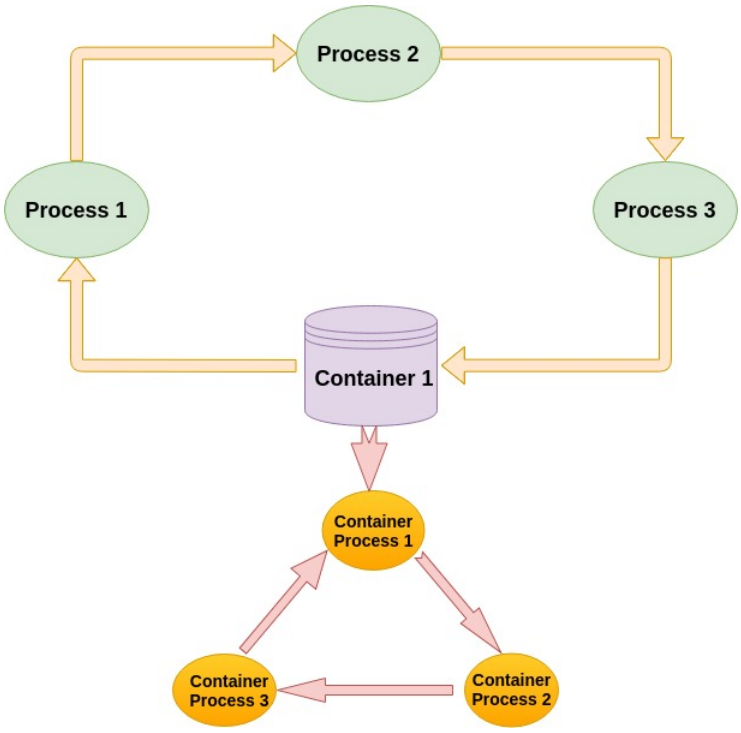

Figure 1: **Round Robin is maintained for both among processes and also among the processes inside a container.**

ps() system call

- This system call prints the name and pids of the active processes, but the process inside a container should only be visible to the processes inside that container.
- On invocation of this call, the kernel first checks the container_id of the process which has made the call.
  - If the call is made by a process which is not in any container, then only the process which are not in any container are printed with their pids.
  - If the call is made by a process which is in a container i, then all the process for which $container\_id = i$ are printed with their pids.

Copy-on-write or COW mechanism

-File System for Containers

- To maintain the file system for each container we need to maintain different copies of a file for each container.
- This is implemented by appending a unique string to the end of each file which uniquely identifies to which container this file belongs.
- Whenever a process inside a container want to open a new or existing file S, a new file is made with name = S + % + *container_id*
- This file is unique to the container and is known to the container with name S only. Whenever the container wants to open the file S using *sys_open*, the container_id of the process is checked and a new file with name S + % + container_id is opened.

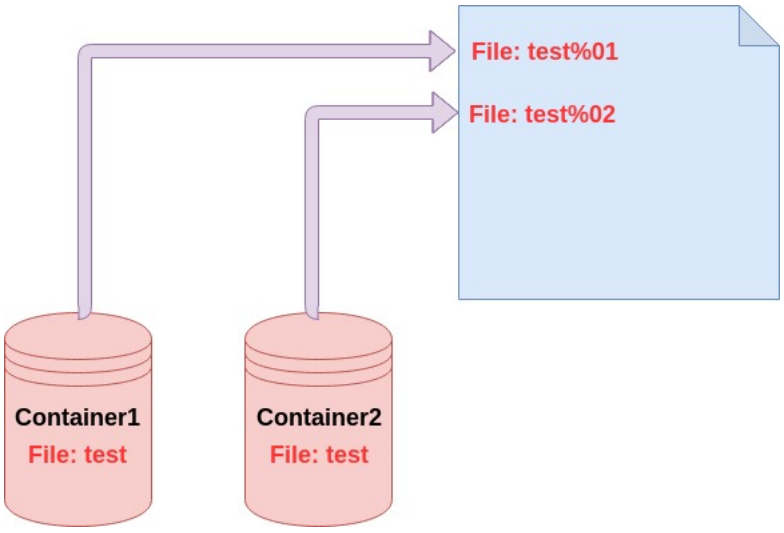

Figure 2: **The files in containers are mapped to actual files by adding a special character and unique identity of the container.**

- This functionality is changed to Copy-on-open instead of Copy-on-write.
- So, whenever a process inside a container want to open a file which is present in the base OS a new file with unique string attached to the name is created and opened. After that the contents of the original file is copied to the new file which is created.

ls() system call

- All the files whose names do not contain the special marker "%" are printed with their names.
- The files whose names contain the special marker "%" are again checked for their *container_id* and if it matches with the *container_id* of the process which has called the function then the name of the file is printed as it is in the same container as the process.