Assignment 3 Report

Yash Malviya 2016CS50403 Ankit Solanki 2016CS50401

Update: April 25, 2019

Abstract

We have implemented containers into xv6. All the data structures and management of them is done in the kernel.

1 Container Data Structures

1.1 Code

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

  int containerId;
  enum procstate vState;
};

// DS for containers
typedef struct procReference {
  int procAlive;  // is proces alive?
  struct proc* pointerToProc;
```

```
} procRef;

typedef struct containerType {
  procRef procReferenceTable[NPROC]; //list of process references in container
  int containerAlive; // is container alive?
  int nextProcRefTableFreeIndex;
  int lastScheduleProcRefTableIndex;
} container;

typedef struct fStructEntry{
  int cfd; // container file descriptor
  int fd; // file descriptor
  int pid; // process id
} fEntries;

#define maxFilesNum 10

struct fTable{
  fEntries fEntry[maxFilesNum];
  int lastFileIndex;
  int nextCfdIndex;
};

#define maxContainerNum 10
```

1.2 Description

1.2.1 PCB

Process Control Block is modified to contain 2 additional things

- vState - contains virtual state i.e state used by the virtual scheduler.
- containerId - Id of container.

1.2.2 Container Process Table

Every container keeps pointers of its processes in the kernel's process table.

- procReferenceTable[NPROC] - Array of tuple of pointer to process(pointerToProc) and whether slot in array is used or not(procAlive).
- containerAlive - maintains state of container (Active or Inactive).
- nextProcRefTableFreeIndex - used to insert item to table.
- lastScheduleProcRefTableIndex - used in containers' virtual scheduling

1.2.3 FS

Array of container struct is stored in the kernel space.

2 Container creation and destruction

Container are defined such that there state is inactive. All information in container is cleared when container is destroyed.

```
int create_container(void);
int destroy_container(int container_id);
```

2.1 create_container

1. Set containerAlive to 1.

No need to reset container before creating as that is done at time of destroying the container. Process Table of container is empty when a new container is created

2.2 destroy_container

1. Obtain containerId and therefore the container from the PCB of process that makes the call.
2. Kill all process alive in the container. Checked from process reference table.
3. Empty the process reference table.
4. FILE SYSTEM
5. Set containerAlive to 0.

Container is reset here so it can be used for future use.

3 Joining and Leaving a container

```
int join_container(int container_id);
int leave_container(void);
```

3.1 create_container

1. Set containerId in process.
2. Save process state into virtual state.
3. Set process state to SLEEPING.
4. Put process the next available slot process reference table of container.
5. Adjust next available slot index.

3.2 leave_container

1. Set containerId to -1 (default, process does not belong to any container).
2. Restore the state of process. Save value of vState to state.

3.3 ps()

1. It shall print all the processes, if not in state UNUSED.
2. Iterate over the kernel's process table.
3. If containerId of process is same as containerId of caller, then print the process name and pid to the console.

4 Virtual Scheduler

Each container has its own virtual scheduler. All virtual schedulers use round robin scheduling algorithm.

1. When round robin for kernel is at the top of the process table, run virtual scheduler for alive containers.
2. Virtual scheduler searches for next process with RUNNABLE vstate.
3. It sets state of the process RUNNABLE, so that it can be scheduled by systems' scheduler in the next round.
4. After the process scheduled by the virtual process finishes running, the state is saved into vState and state is set to sleeping again.

5 Virtual File System

- Each container saves the unique id of file nodes that have been opened by any process encapsulated in that container. This directly implies that any file opened in an container will be visible to the entire container, and it is implemented as so.
- Each container sees the file system of host, and the files opened by its processes.
- If a process inside a container tries to open the file in host system, it first copies it into container.

5.1 ls() call

Opening a file named $textfile.txt$ by process1, where same name file is already present in the system shall print an output like: activate=false

```
filename: .              | ID: 1
filename: ..             | ID: 1
filename: README         | ID: 2
```

```
filename: cat            | ID: 3
filename: echo           | ID: 4
filename: forktest       | ID: 5
filename: grep           | ID: 6
filename: init           | ID: 7
filename: kill           | ID: 8
filename: ln             | ID: 9
filename: ls             | ID: 10
filename: mkdir          | ID: 11
filename: rm             | ID: 12
filename: sh             | ID: 13
filename: stressfs       | ID: 14
filename: usertests      | ID: 15
filename: wc             | ID: 16
filename: zombie         | ID: 17
filename: test           | ID: 18
filename: test2          | ID: 19
filename: test1          | ID: 20
filename: console        | ID: 21
filename: testfile.txt   | ID: 22
filename: testfile.txt | ID: 23 | CID: 2
```