Udit Jain
2016CS10327
COL 331 : OS

Assignment 3 Report

Shreshth Tuli
2016CS10680
April 25, 2019

Container Manager

In this section, we had to implement a container manager data structure which will be in charge of all the containers and will be modified by system calls to create, read, update and delete the containers respectively.

Container manager has the following data structure for generality and storing logs:

```
containerStruct containerManager = {
  .containerIDs = { 0 },
  .numActive = 0,
  .procIDs = {{ 0 }},
  .notAllowed = {{ 0 }},
  .systemCalls = {{ 0 }}
};
```

Algorithm 1: Container Manager

Where container id's is the id's of the current containers, procId's are the processes running in the respective containers, not-allowed are the system calls not allowed in that container, because every process's system call has to pass verification before initiation, and systemCalls are the system calls executed by processes inside. (To ensure that container manager can keep track of system calls and other data realted to container).

System calls were created with the following signature:-

- uint create_container(void)
  It creates a new container with the default parameters from the container manager.
- uint destroy_container(uint container_id)
  It destroys the specified container and handles clean-up actions like deleting the files and clearing out memory and processes of that container.
- uint join_container(uint container_id)
  It is used by a process to join a particular existing container. Implemented using fork command. Scheduled by the virtual scheduler later
- uint leave_container(void)
  It is used by a process to leave the container and go back to the host container (by default) or be cleared.

```
// MOD-3 : Container syscalls
#define SYS_create_container    37
#define SYS_destroy_container   38
#define SYS_join_container      39
```

```
#define SYS_leave_container      40
#define SYS_proc_stat_container  41
#define SYS_scheduler_log_on     42
#define SYS_scheduler_log_off    43
#define SYS_cid                  44
#define SYS_memory_log_on        45
#define SYS_memory_log_off       46
#define SYS_memory_gva           47
```

Algorithm 2: System Calls added

Virtual Scheduler

As we are mimicking the Container behaviour, we have the following mechanism for correctly scheduling the processes of a container. We have edited the Process Control block to include the container information in the PCB itself.
The default scheduler in xv6 is modified to take into account the PCB's container id field and accordingly schedule the processes. We are still following the Round Robin policy.

The new process Control block is :

```
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 trapframe *Oldtf;      // MOD-1 : Old trap frame
  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)
  sig_handler sig_handler;      // MOD-1 : Signal handler for process
  char msg[message_size];       // MOD-1 : Signal msg
  int disableSignals;           // MOD-1 : Disable signals when currently ↩
      processing one
  int interrupt;
  int containerID;              // MOD-3 : Container ID
  enum v_procstate v_state;     // MOD-3 : Virtual process state
};
```

Algorithm 3: Process Control Block

And it is scheduled in the following manner :-

```
p->v_state = V_RUNNING; // MOD-3 : Make this process virtually running
  // MOD-3 : Print schedule if log on
  if(container.containerIDs[p->containerID] == 2)
    cprintf("Container:%d\tScheduling process:%d\n", p->containerID, p->pid);
  // Other process which was v_running in this container to shift to v_runnable
  for(p1 = ptable.proc; p1 < &ptable.proc[NPROC]; p1++){
    if(p1->containerID != -1 && p1->containerID == p->containerID && p1->v_state == V_RUNNING && p1->pid != p->pid){
    // checking the schedule status here and virtual scheduling
      p1->v_state = V_RUNNABLE;
    }
  }
```

Algorithm 4: Additions in Scheduler

System Calls Handling

There were many ways to handle the invocation of system calls and their execution such that they are logically separate in different contianers and were logged to the containers and the respective container could keep track of the system call invoked from it's virtualized processes.

The second challenge here was to keep certain system calls being allowed and certain not allowed from a particular container. Verification of system calls as metioned in the document.

We managed both the expectations by maintaing a table of not-allwed system calls in the container itself and changed the code of the default syscall(id) function itself to check for the appropriate parameters.
The code change's required was in the container manager for storing the not-allowed calls which is shown in Container Manager data structure. Other was in syscall invocation function, the checking for allowed and not allowed system calls:

```
for(int j = 0; j < 100; j++){
  if(container.notAllowed[curproc->containerID][j] == num)
    cprintf("Syscall %d not allowed in container with ID %d\n",
        num, curproc->containerID);; // syscall not allowed from this container
}
```

Algorithm 5: Syscalls Changes

Resource Isolation

We've implemented both parts of this, the page table address translation and the virtual file system.

For the first part, we are keeping a lookup of the addresses used by different processes in the container and thus calculating the GVA, HVA. Calling this function from the changed malloc function. Here are the necessary code changes. :

```
int sys_memory_gva(int size, int hva)
{
  argint(0, &size);
  argint(1, &hva);
  int id = myproc()->containerID;
  int addr = container.gva[id];
  container.gva[id] = addr + size;
  if(container.memorylog[id] == 1)
    cprintf("GVA : %d, HVA : %x\n", addr, hva);
  return addr;
}
```

Algorithm 6: Address calculation and translation

For the 2nd part in the virtual file system our model is to create directories for different containers and keep them in isolation, which also provides independence for copy on write.

Our ls prints all the host's files and not the container's directories (by design). When ls through container, due to COW, it shows host's file but on opening/read/write it deals with the local file. This is done in this way because we want host file to remain visible. The relevant code is in split into different functions in the file and too big to include the core part here. When we're opening file for read, we check if the file is in container, if not then pick host file. For write we check if it's in container directory, if it's there, then write at the end of that file. If not, then check it's in the host, then copy the host file to local directory of container and start writing to it.

```
...
if(fd < 0 && mode & O_RDWR){ // Not in container and write -> copy to my container
    fd = open(name, O_CREATE|O_RDWR);
    int fd_host = open(filename, O_RDONLY);
    while((n = read(fd_host, buf, sizeof(buf))) > 0){
      write(fd, buf, n);
    }
  }
...
```

Algorithm 7: Copy on write mechanism

Testing

We tested with two testing scripts in the user level program, container_test.c and cts.c .

In the script cts, a container is joined and a file is there, file1, and when you leave the container, the file1 which opens the host version. This among other function like COW are tested in this script. The script and output are given the the folder.

These scripts and their outputs are given in the root directory.

- ls() : Displaying files correctly, detailed output given in folder.
- ps() : Running from container_test script so showing it's name in the process name column.

```
Init 4
Child proc 4, num 1
Init 5
Init 6
Child proInit 7
Init 8
Pid     Name            MemSize Killed  State
1       init            12288   0       SLEEPING
2       sh              16384   0       SLEEPING
3       container_test          16384   0       RUNNING
4       container_test          16384   0       SLEEPING
5       container_test          16384   0       RUNNING
6       container_test          16384   0       RUNNABLE
7       container_test          16384   0       RUNNABLE
8       container_test          16384   0       RUNNABLE
c 6, num 3
Child proc 7, num 4
Child proc 8, num 5
CParent proc 3, num 0
hild proc 5, num 2
pid:4   name:container_test     state:WAITING
pid:5   name:container_test     state:WAITING
pid:6   name:container_test     state:RUNNING
pid:7   name:container_test     state:RUNNING
pid:8   name:container_test     state:RUNNING
Container:0     Scheduling process:5
... (Other schedules)
Container:0     Scheduling process:4
```

Algorithm 8: ps output on xv6