Mohit Gupta : 2016CS50433
Shashwat Banchhor:2016CS10350
Assignment 3
COL331
April 20, 2019

Implementation Methodology for all Parts of the Assignment

Part 1: Standard Operations:

1. **Base Data Structures:**
   - We have made a **container table**(struct) which contains the list of all the container with the limit of maximum 1024 containers.
   - We then created a **container lock** for accessing the container table.
   - Each **container** is a struct containing it's unique container id as well as the information about the page table, process table and files table created by it's processes.
   - **Process table** is maintained by assigning each process the container id it belongs to(added new variable *in_container_id* in struct proc).
   - **File table** is maintained by each container by assigning each file the container id it was created by(added new variable *cid* in inode struct as well as dinode struct).
   - **Page table** is maintained by each container which keeps track of the memory(GVA) allocated by it's processes(added new variable in *container* struct) assuming GPA = HVA.
2. **Part 1.1 : Create Container :** We acquired the container lock and then traversed the *container_table* to find a free slot, and if found we filled the slot with the unique input id, released the lock and returned non-negative integer.
3. **Part 1.2 : Join Container :** We acquired the container lock and then traversed the *container_table* to find a appropriate container the process wants to join.If the container the process wants to join isn't valid then we return -1 otherwise we set the process container id and return non-negative integer. We have also ensured that a process isn't already in a container when it calls the join system call.
4. **Part 1.3 : Leave Container :** We set the process container id to -1.
5. **Part 1.4 : Destroy Container :** We acquired the container lock and then traverse the *container_table* and delete the appropriate container slot and returned a non-negative integer. Every process needs to be killed. Also, all the mapping tables and any files created within that container are discarded from now on as they have an invalid cid.

Part 2 Command Output:

Virtual File System Implementation:

- The inode for every file has a variable to maintain it's container id and file table.
- Whenever a file is created the inode's id is set to the process's container id.
- To ensure **File Resource Isolation for creation as well as editing**, whenever a file is requested to be read/write, the appropriate file with the inode-¿cid equal to process's container id is opened and edited.
- Also, all the files that were initially present before in the file table are accessible by all containers.
- To assure **file data consistency between RAM and disk**, whenever a file is closed, the meta-data of the file is appropriately switched to the dinode structure maintaining the container id too.
- Hence for a container it's file table is all the files that have their container id set to the id of the container.

```
// in-memory copy of an inode
struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  struct sleeplock lock; // protects everything below here
  int valid;          // inode has been read from disk?

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
  // ADDED
  int cid;
  int path_len;
};
```

```
// On-disk inode structure
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses


  // ADDED
  int cid;
  int path_len;
```

```
// Lock the given inode.
// Reads the inode from disk if necessary.
void
ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
    panic("ilock");

  acquiresleep(&ip->lock);

  if(ip->valid == 0){
    // cprintf("-----------------------------------
    bp = bread(ip->dev, IBLOCK(ip->inum, sb));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    ip->cid = dip->cid;
    ip->path_len = dip->path_len;
```

```
// Copy a modified in-memory inode to disk.
// Must be called after every change to an ip->xxx field
// that lives on disk, since i-node cache is write-through.
// Caller must hold ip->lock.
void
iupdate(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  bp = bread(ip->dev, IBLOCK(ip->inum, sb));
  dip = (struct dinode*)bp->data + ip->inum%IPB;
  dip->type = ip->type;
  dip->major = ip->major;
  dip->minor = ip->minor;
  dip->nlink = ip->nlink;
  dip->size = ip->size;
  dip->cid = ip->cid;
  dip->path_len = ip->path_len;
```

```c
struct stat {
  short type;  // Type of file
  int dev;     // File system's disk device
  uint ino;    // Inode number
  short nlink; // Number of links to file
  uint size;   // Size of file in bytes
 //
  int cid;
  int path_len;
};
```

```c
// Copy stat information from inode.
// Caller must hold ip->lock.
void
stati(struct inode *ip, struct stat *st)
{
  st->dev = ip->dev;
  st->ino = ip->inum;
  st->type = ip->type;
  st->nlink = ip->nlink;
  st->size = ip->size;

  // ADDED
  st->cid = ip->cid;
  st->path_len = ip->path_len;
}
```

1. **Part 2.1 ls command:**
   - The ls system call respects the isolation provided by the containers by reading the file table and prints only the input file entries which have their container id same as the container id of the calling process. Each container appends it's container id to the file name of any new file that it creates(Please refer COW(Part 2.5) section for details). The file names outputted is truncated by the file's cid. This is to provide consistency with how we create the new files(Please refer COW(Part 2.5) section for details).
   - Also, the files initially present in the file table, which are global to all containers are also printed.
   - Thus, the files created by a process on container B is not printed in the output of ls command issued by container A, when A $\neq$ B.
2. **Part 2.2 : ps command:**
   - We acquire the process table and print all the current RUNNING or WAITING process from the process queue belonging to the container.
   - Thus, the process started (joined) within a container the same container will be printed in the ps called by processes inside that container only and not to other processes running in some other container.
3. **Part 2.3 : Scheduling the processes:**
   - We have implemented round-robin scheduling in the scope of containers.
   - We traverse the container table in a cyclic manner, to ensure that every container is scheduled at least once ensuring starvation freedom.

- We schedule the next process belonging to a different container or a different process in same container than previously scheduled in the table(traversed cyclically) to ensure every process inside a container is also scheduled at-least once.
- We also implemented the *scheduler_log_on* and *scheduler_log_off* system calls which toggles a global variable, and prints the scheduled process cids and pids when the variable is 1.

4. **Part 2.4 : Page Table(container malloc command):**
   - We implemented the *memory_log_on* and *memory_log_off* system calls which toggle a global variable *mem_log*.
   - We implemented a *container_malloc* user process which when called by any user process, first malloc's the requested number of bytes and then updates the page table of the current process's container by making a system call *container_malloc_syscall*.
   - When *mem_log* is 1, any new mappings added to the table of any container are printed.
5. **Part 2.5 : Copy-on-write or COW mechanism:**
   (a) **Part 2.5.1 : File isolation among containers for newly created files:**
      - **A file created by a process in container A is not accessible to any process outside the container.** We implemented this by making making changes in the open system call. Basically, we had already maintained the inode-¿cid for every file created, so we just check whether the process's container id matches with the file's container id. If it happens, then file requested is made accessible else not.
      - If a mapping is already present with the same file name but different container id, a new file is created and to keep the name of files different in the file table, we append the original file name with container id of process and use this appended name in the file table.
      - **Two files of the same name created by two different process in different containers should be different.** In the open system call, we check if the file is already created with the same name. If yes, we then check it's container id, if it's different from the calling process's container id then we create a new file and add it to the file table with the current process's container id. Thus, this ensures that two different mappings are created in the table and so, on access too, different files will be accessed by different containers.

   (b) **Part 2.5.2 : File isolation among containers for already existing files:**
      - A new copy of the existing file is created by the container(by using read and write system calls) as soon as a process in the container starts to modify the

file and the container id of the new file created is set to the id of the container and this new mapping is added to the file table.

- To keep the name of the new copy file different, we append the original file name with container id of process and use this appended name in the file table.
- This ensures that as soon as a processes starts to modify the file,the changes made to the file are local to the container. and not visible outside the container. Other containers don't see the changes in this new file as the file's inode-¿cid is different and also, when they search for a file with same name, they access the mapping in the file table corresponding to the global file.

6. **Extra Credits:**

   (a) We have handled the cases where in it is not necessary for the process to reside in a container while there are some processes inside containers. That is, we maintain the resource isolation as well as global scopes even if there is a mixture of processes, some of which are in some container while others are global.

   (b) We have implemented **container_malloc** and the page table in the user space. This allows the returned address to be in the container's virtual space(instead of kalloc's address in kernel) and shifts the maintenance of memory page tables to user space.

   (c) We have also imposed the correct added behaviour of container join and leave by a process i.e. a process should leave it's current container(if any) before wanting to join any other container.