COL331 - Operating Systems
Assignment 3

March 15, 2020

Nilaksh Agarwal - 2015PH10813

Saransh Verma - 2016CS10326

Objective

To impliment a container in xv6 to support virtualization:

1. **Container manager**: A user program which will mimic the functionalities provided by a container service.
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
3. **System calls**: 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.
4. **Resource isolation**: Containers are isolated from each other by using the mechanisms provided by the kernel.

1 Container Manager

In this part, we created system calls, and a container structure to manage the container creation, allocation, etc. and also implemented a basic round-robin scheduler within the container.

```
        struct container{
        int container_id;
        int created;
        int process;
        int p_list[MAX_PROCESS];
        struct proc* proc_file[MAX_PROCESS];
};
```

Here the container_id is a unique value given to each container. The created flag is a binary value to indicate if that container has been created or not. The process variable stores the number of processes linked to this container. Finally the p_list and proc_file stores the *pid* and proc structure of each process linked to this container.

Container Creation

After this, we implemented system calls for creating and destroying containers. This was done through the created flag, which tells us if that container is active or not.

```
int create_container(void) {
  if(container_init==0) {
    init_container();
    container_init = 1;
  }
  int i;
  for(i=0; i<MAX_CONTAINERS; ++i) {
    if(cont[i].created == 0) {
      cont[i].created = 1;
      return cont[i].container_id;
.....

int destroy_container(int id) {
  for(int i=0; i<MAX_CONTAINERS; ++i) {
    if(cont[i].container_id == id && cont[i].created == 1) {
      cont[i].created = 0;
      for(int j=0; j<MAX_PROCESS;++j) {
        struct proc* c_proc = cont[i].proc_file[j];
        c_proc->c_id = 0;
      }
.....
```

Here, we first initialize the container if we haven't already done it before, checking the container_init flag. After that we search and find an unallocated container, and create it. Similarly, to destroy a container, we free all processes in it, the delete the container.

Joining a container

We implemented two system calls for joining and leaving a container respectively, which will link a particular process to that container. This is implemented by adding the process' pid and proc structure to the container, and increasing the number of processes by one, and vice versa.

```
int join_container(int id) {
  struct proc* cur_proc = myproc();
  cur_proc->c_id = id;
  for(int i=0; i<MAX_CONTAINERS;++i) {
    if(cont[i].container_id == id && cont[i].created == 1) {
      for(int j=0; j<MAX_PROCESS; ++j) {
        if(cont[i].p_list[j]==0) {
          cont[i].p_list[j] = 1;
          cont[i].process += 1;
          cont[i].proc_file[j] = cur_proc;
          return 1;
          ......

int leave_container() {
  struct proc* cur_proc = myproc();
  int c_id = cur_proc->c_id;
  for(int i=0; i<MAX_CONTAINERS;++i) {
    if(cont[i].container_id == c_id && cont[i].created == 1) {
      for(int j=0; j<MAX_PROCESS; ++j) {
        if(cont[i].proc_file[j]== cur_proc && cont[i].p_list[j] == 1)
            {
          cont[i].p_list[j] = 0;
          cont[i].process -= 1;
          return 1;
          ......
```

Virtual Scheduler

Here, we add a container_state to the process structure, which we use to schedule processes within the container. We also add the container ID to this structure.

```
enum contstate {BUSY, READY};

struct proc {
  ..........
  enum contstate c_state;        // Container State
  ..........
  int c_id;                     // Container ID
};
```

In the scheduler function, we employ a round robin selection of processes to run, from all the processes within the container. Selecting this process, we change it's container_state to "READY". Also, while selecting a process to run, we check it's container state as well, but only if it's a part of a container.

```
void scheduler(void) {
.....
```

```
count++;
    for(int i = 0;i<MAX_CONTAINERS;i++){
      if(cont[i].created == 1) {
        for(int j=0; j<MAX_PROCESS; ++j) {
          if(cont[i].p_list[(j+count)%MAX_PROCESS]!=0) {
            if(cont[i].proc_file[(j+count)%MAX_PROCESS]->state ==
               RUNNABLE) {
              cont[i].proc_file[(j+count)%MAX_PROCESS]->c_state =
                 READY;
              break;
.......
if(p->state != RUNNABLE || ((p->c_state != READY) && (p->c_id != 0)))
        continue;
.......
```

Directory Structure and CoW

Directory is a file which contains a sequence of struct DIRENT(Directory Entry), we edited this struct to include information regarding which containers can see this entry.

```
struct dirent {
  ushort inum;
  char name[DIRSIZ];
  ushort containersAcc[8]; // if containersAcc[i] == 0 then this
     entry is visible in ith container, container 0 indicates root
};
```

Now there are two important functions:

```
int
dirlink(struct inode *dp, char *name, uint inum); // this functions
   adds a new entry to the directory

struct inode*
dirlookup(struct inode *dp, char *name, uint *poff); // this function
    checks for a file of name in the directory
```

So if a container creates a file of name fn, other containers should not be able to access this file, this is ensured by dirlookup, I added one more condition that the DIRENT of that name should also be accessible for that container.

```
struct inode*
dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  struct proc *curProc = myproc();
  // ushort containerNum = curProc->
  ushort c_id = curProc->c_id ;

  for(off = 0; off < dp->size; off += sizeof(de)){
```

```
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlookup read");
    if(de.inum == 0)
      continue;
    if(de.containersAcc[c_id] !=0) // New Condition
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}
```

Changes in dirlink are made to facilitate CoW, so lets first see the logic used for CoW and then I will explain the changes in dirlink. For CoW I will store two 2d arrays:

```
char filenames[MAXPROC][NOFILE][NAMESIZE]; // stores the names of the
    file indexed by process and file descriptor
ushort lastOpeningContainer[MAXPROC][NOFILE]; // stores which
   container last edited the file index by process and file
   descriptor
```

Now first we'll see CoW and then dirlink, but for now lets assume the following about dirlink:

If we want to add a DIRENT, and a DIRENT of the same name exits, then it will make that DIRENT inaccessible for the current container and create a new DIRENT which will only be accessible for this container.

And another function in openFile which is same as sys open except it takes path and mode as arguments.

```
int openFile(char *pathArg, int omode){
  char path[NAMESIZE];
  int fd, omode;
  struct file *f;
  struct inode *ip;
  int pid = myproc()->pid;
  strncpy(path,pathArg,NAMESIZE);
    begin_op();

  if(omode & O_CREATE){
    ip = create(path, T_FILE, 0, 0);
    if(ip == 0){
      end_op();
      return -1;
    }
  } else {
    if((ip = namei(path)) == 0){
      end_op();
      return -1;
```

```
    }
    ilock(ip);
    if(ip->type == T_DIR && omode != O_RDONLY){
      iunlockput(ip);
      end_op();
      return -1;
    }
  }

  if((f = filealloc()) == 0 || (fd = fdalloc(f)) < 0){
    if(f)
      fileclose(f);
    iunlockput(ip);
    end_op();
    return -1;
  }
  iunlock(ip);
  end_op();

  f->type = FD_INODE;
  f->ip = ip;
  f->off = 0;
  f->readable = !(omode & O_WRONLY);
  f->writable = (omode & O_WRONLY) || (omode & O_RDWR);
  strncpy(filenames[pid][fd],path,NAMESIZE);
  return fd;


}
```

so the new sys write becomes as follows

```
int
sys_write(void)
{
  struct file *f;
  int n;
  char *p;
  int fd ;
  argint(0,&fd);

  int cid = myproc()->c_id;
  int pid = myproc()->pid;
  if(lastOpeningContainer[pid][fd] == cid){ // here we are checking
     whether the last container which edited the file is our current
      container

// If true then nothing is changed
    if(argfd(0, 0, &f) < 0 || argint(2, &n) < 0 || argptr(1, &p, n) <
        0)
      return -1;
    return filewrite(f, p, n);
  }

  else
  {
```

```
  // if not then we create a new file with same name, dirlink will do
       its function as described abouve
    lastOpeningContainer[pid][fd] = cid;
    char *fileName ;
    strncpy(fileName,filenames[pid][fd],NAMESIZE);
    int fdNew = openFile(fileName,O_CREATE);
    struct file *oldFile = myproc()->ofile[fd];
    struct file *newFile = myproc()->ofile[fdNew];

  // we copy contents of the old file
    char *p2;
    while(fileread(oldFile,p2,1)){
      filewrite(newFile,p2,1);
    }
  // we assign new file to the file descriptor and write the original
       text on this new file
    myproc()->ofile[fd] = newFile;
    return filewrite(newFile,p,n);
  }

}
```

Now the dirlink function.

```
[language = C]
int
dirlink(struct inode *dp, char *name, uint inum)
{
  int off;
  struct dirent de;
  // struct dirent de
  struct inode *ip;
  struct proc *curProc = myproc();
  // ushort containerNum = curProc->
  ushort c_id = curProc->c_id ;

  // Check that name is not present.
  if((ip = dirlookup(dp, name, 0)) != 0){
    iput(ip);
    return -1;
  }

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(namecmp(name,de.name) == 0){ // finds DIRENT of same name and
        makes it inaccessible
      de.containersAcc[c_id] = 1;
      if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
        panic("dirlink2");
    }

  }

  // Look for an empty dirent.
  for(off = 0; off < dp->size; off += sizeof(de)){
```

```
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      break;

  }

  strncpy(de.name, name, DIRSIZ);
  de.inum = inum;
  int i;
  for(i = 0 ; i < 8 ; i +=1){  // New DIRENT only accessible for our
      current container
    de.containersAcc[i] = 1;
  }
  de.containersAcc[c_id] = 0 ;

  if(writei(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
    panic("dirlink");



  return 0;
}
```