Project Report

Junyi Xiao
Duke University
Durham, USA
junyi.xiao@duke.edu

Hao Jiang
Duke University
Durham, USA
hao.jiang@duke.edu

ABSTRACT

For CS510 final project, we add some system calls to interact with users and implement some container functionality for xv6, a simple, Unix-like teaching operating system. QEMU enables us to run xv6 locally. For this project, we've decided to build on our feature addition on xv6 since it's a reimplementation of for an old version of Linux. Our container isolates PID and file system based on namespace. Unlike Linux, where PID namespace is decoupled from file system namespace, we wrap them together in the container.

KEYWORDS

xv6, namespace

ACM Reference Format:
Junyi Xiao and Hao Jiang. 2020. Project Report. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

For our final project, we want to choose a topic where we have some experience, so that we could possibly imitate the real effect. With docker used in every projects this semester, we've decided to implement the basic functionality of PID and file system namespace isolation.

The rest of this report is organized as follows. Section2 describes our inspection into xv6, and we adapt the way of user interaction. Section3 gives an overview of our design for the project, and the detailed implementation, including PID isolation and file system isolation. Section4 briefly shows how we do testing. Section5 includes PID and file system isolation demonstration. Section6 mentions our next step to prospect and some known bugs. Section7 talks about the lesson learnt in this project. Section8 gives a final conclusion.

2 XV6 INSPECTION

2.1 Key points in xv6

xv6 is a simple, Unix-like Operating System developed at MIT during the summer of 2006. It's a reimplemenation of Unix Version6 which was developed in the 1970s. We select xv6 as our development base code because it's similar to Linux OS, which we use every single day, and its source code is not that complicated. For this project, we focus on process of xv6(related files: proc.h, proc.c, sysproc.c, sh.c). One of the challenges is since xv6 is a bare bones OS, it lacks lots of tools and features that we used to have in standard Linux OS. So as first part, we plan to start our project by adding a few user programs and their corresponding system calls, especially those could easily show OS status, say, ps and pwd.

bin directory In Linux, we could execute system executable like cat, link everywhere, and we don't need to copy or move them into current directory for execution.

In Linux, bin direcctory includes essential ready-to-run programs (binaries), includes the most basic commands such as ls and cp. In this project, we also created a bin directory in mkfs stage, and moves all binary executable user program into that directory.

Apart from that, all user-level programs are run via exec. Originally it just loads program under current directory into memory. But we change it to locate the program from bin directory first, so that all user-level programs can be executed everywhere.

pwd This is a frequently-used command in Linux for showing current working directory of the running process, but it's not available in xv6. Either for debugging, and demonstrating the result of file system isolation, the knowledge on current working directory is needed.

For file system isolation, specifically, instead of the absolute path of the whole operating system, path shown on the standard output should be the absolute path inside of the running container.

ps This is also a widely-used util command in Linux, used to printing all alive processes along with their PID, process name. After a little research into proc.c, we found it has already provided a function named procdump with the similar functionality.

For this project, ps should also display container status, including container name, CID, processes alive inside. Our task is to add a syscall, and do little adaption to procdump.

3 PROJECT DESIGN AND IMPLEMENTATION

3.1 Adding util commands

Add user-level commands in xv6 is not that hard, but it involveds several files and stages. We first researched how traps are handled in xv6, then register new syscall and corresponding upper-level API. The detailed implementation includes: (1) define function at defs.h (2) register syscall at usys.S (3) append syscall id at sys_call.h (4) define extern function, and register syscall in the syscall function pointer table (5) implement the detailed function at sysproc.c or sysfile.c

3.2 Design a data structure of container

Based on the intention to provide users the flexibility of creating, starting, pausing, resuming and stopping container, we've designed the user-level commands as table 1.

Table 1: Container Commands

Commands	Functionality
cont create <cont name>	Create a container
cont start <cont name> prog [arg..]	Start container, run program
cont pause <cont name>	Pause the container
cont resume <cont name>	Resume the container
cont stop <cont name>	Stop container and processes inside

Notice that we use cont as the general command prefix, with different following trailing part indicating specific purposes.

From a high-level, we need a data structure to represent the status of a container, just like process. And also a table to hold all containers, just like ptable. So we define them as follows:

```
enum contstate {
  CUNUSED,
  CEMBRYO,
  CREADY,
  CRUNNABLE,
  CRUNNING,
  CPAUSED,
  CSTOPPING
};

struct container {
  // Container ID
  int cid;
  // Root directory
  struct inode *rootdir;
  // Root directory in string
  char rootpath[200];
  // Container state
  enum contstate state;
  // Table of processes owned by container
  struct proc *ptable;
  // Next process to schedule
  int nextproc;
  // Number of processes
  int nproc;
  // Container name (debugging)
  char name[16];
};

struct
{
  struct spinlock lock;
  struct container cont[NCONT];
} ctable;
```

To accomodate the addition of container, we also change the ptable.

```
struct {
  struct spinlock lock;
  struct proc proc[NCONT][NPROC];
} ptable;
```

After defining the data structure for container, multiple process-related functions would be re-organized in the unit of container. It involves several stages: initialization part, scheduling part and exiting part.

(1) The initialization includes initialize the lock of ctable, allocating the space of container, initialize processes inside of the corresponding container.

```
void cinit(void)
{
  initlock(&ptable.lock, "ptable");
  initlock(&ctable.lock, "ctable");
}
```

(2) The main body of scheduling part is scheduler() function, and for exiting part it's wait(), which clears zombie processes, they originally iterate through all processes, now switch to all processes inside of all containers. The whole process can be represent as following.

```
(1) The scheduler ensures the processes scheduled
have to be in a running or runnable container.
(2) The nextproc data member in container
structure ensures processes could be
scheduled and executed in turn.
(3) Update container and process's
status to RUNNING.
(4) Reset container's status. There're
several cases here:
If the container has been paused, it
can only be runnable by cresume().
If the container has been stopped, it
should be cleared by wait().
Otherwise, reset its status to CRUNNABLE.
```

The detailed implementation is:

```
for(int ii = 0; ii < NCONT; ++ii) {
      // Only schedule for running and
      // runnable container.
      cont = &ctable.cont[ii];
      if (cont->state != CRUNNABLE &&
      cont->state != CRUNNING) {
        continue;
      }

      // Loop over the processes of ptable
      // looking for runnable process.
      // And the process should be RUNNABLE.
      p = &cont->ptable[(cont->nextproc++) % NPROC];
      if (p->state != RUNNABLE) {
        continue;
      }
```

```
    // Switch to chosen process.  It is
    // the process's job to release ptable.
    // lock and then reacquire it
    // before jumping back to us.
    c->proc = p;
    switchuvm(p);
    p->state = RUNNING;
    cont->state = CRUNNING;

    swtch(&(c->scheduler), p->context);
    switchkvm();

    // Process is done running for now.
    // It should have changed its p->state
    // before coming back.
    c->proc = 0;
    if (cont->state != CSTOPPING &&
      cont->state != CPAUSED) {
      cont->state = CRUNNABLE;
    }
```

3.3 PID and File system isolation

For PID and file system, every container, as shown above, stores its own data member. So our xv6 should have a way to display that.

For PID isolation, there should be a way to show which process belongs to which container, and display real PID and relative PID within container. ps command calls does ctable status fetching and printing via trap.

While for file system isolation, our intuition is to get full path for the OS, then root directory of the container, the difference between them is the the directory inside of container. The way I get the full path of the OS is to search up-level directory, until the root directory '/' is reached, and it's determined by inode number checking(stored at struct dirent). Another path needed is the root directory of currently running container, it's stored at the creation step, so it's immutable.

Relative two user commands are pwd and cd. The implementation of pwd can be represented at following code.

```
(1) get current working directory by
searching upward till root
(2) fetch root directory of running
container via trap
(3) check whether container root is the
path prefix of cwd, and fill their
difference into buffer and pass back
to user space
```

For cd, there're two key functions. One is sys_chdir(at sysfile.c), the other is main(at sh.c). We used to consider directly adapting chdir, but it's under syscall level, and only kernel-level functions could be of help. Since kernel functions are usually using struct inode, it needs another translation from inode to path. So we select the second method: implementing the first half in the shell.

It involves two stages: sh program reads user input and interpret, and calls sys_chdir. cd's responsibility involves:

(1) interpret and check the user-input text, this primarily needs resolution of three special cases: '.' and '..', which represents "stay at the current directory" and "go back to the upper-level directory if not root".

(2) due to the attribute of file system isolation, path-switching have to be within the scope of container, so before actually calling sys_chdir, validity of the target path has to be checked.

(3) after parsing and checking, sys_chdir is called and current working directory for the running process is updated.

3.4 User-level Program

Apart from container data structure inside of kernel, we also provide succinct user programs. **cont create <cont_name>** This command is used to create a container inside kernal. It should allocate space in kernel's ctable, set the container's status to CREADY and indicate user whether creation is successful. If fail, an error message should display the reason.

cont start <cont_name> prog [arg..] This command is used to start a container and set its status to CRUNNING. Alongwith the command as a program, it also execute the command inside of the container. At any moment, there should be only one container running. The default running container is the root one, which is started when the OS launches. If fail, an error message should display the reason. One thing need to note, when trying to start the container, the current working directory have to be within the scope of container's root directory.

cont pause <cont_name> This command is used to pause a container and set its status to CPAUSED. It's valid only when the container is running or runnable. If fail, an error message should display the reason.

cont pause <cont_name> This command is used to resume a container and set its status to CRUNNABLE. It's valid only when the container has been paused beforehand. If fail, an error message should display the reason.

cont stop <cont_name> This command is used to stop a container and set its status to CSTOPPING. At the stopping period, the kernel is responsible for freeing the resources allocated before, and stop all processes inside by setting their status as ZOMBIE and transfer them to initproc and root container. If fail, an error message should display the reason.

3.5 How we stop a container

Stopping containers is the main challenge we've encountered in this project. Based on our knowledge on container, every container has a initprocess, which is the parent process for all spawned processes, especially for adopting and exiting zombie processes.

```
root@c70b3c5c83e5:/cps510/final_proj/xv6# ps
  PID TTY          TIME CMD
    1 pts/0    00:00:01 bash
75710 pts/0    00:00:00 ps
```

Our method is we don't explicitly create a initproc when the container is created, instead we assign the processes stopped and exited in self-created container the parent container as root container, and parent process initproc, so that it will handled the exiting of these processes, just as normal one when there're no containers.

4 TESTING

Unit Testing usually requires testfiles which aims at testing one single function or one single class. But this kind of hard to apply to Operating system. For this project, we mainly apply manual testing.

Every time we've made changes to the source code, we did testing by repeated mkdir, ps, ls and pwd, validating whether it meets our expectation.

Also, for some functions which is quite isolated and xv6-independent, we also edit some test cases. For example, for path concatenation and filtering, we've created a test file "path_concatenation_test.c".

5 NAMESPACE REAL EFFECT DEMONSTRATION

5.1 pwd at the initial working directory

```
$ pwd
Full path is /
Current running directory is /
/
```

It will print the full path, container root directory and relative path seperately.

5.2 ps at the initial OS state

```
$ ps
Container 1 : root container running ,
root path = /
Process         PID      Real PID
Status          Container
init             1        1
sleep           root container
sh               2        2
sleep           root container
ps               4        4
running         root container
```

5.3 create a container

```
$ mkdir test_dir
$ cd test_dir
Ready to call sys_chdir...
Full path = /test_dir
Container's rootdir = /
$ cont create test_cont1
Current working directory is
/test_dir
full path is : /test_dir/test_cont1
directory /test_dir/test_cont1
has been created successfully
Container test_cont1 created at
/test_dir/test_cont1 successfully.
```

5.4 start a container outside of its scope

```
$ pwd
Full path is /test_dir
Current running directory is /
test_dir
$ cont start test_cont1 pwd
Current working directory is
/test_dir
Root directory for container
test_cont1 is /test_dir/test_cont1
Error, starting container has to
be in its root directory
```

5.5 start a container successfully, and inspect via ps

```
$ cont start test_cont1 ps
Current working directory is
/test_dir/test_cont1
Root directory for container
test_cont1 is /test_dir/test_cont1
Start container test_cont1
with cid 2 succeeds.
$
Container 1 : root container runnable,
root path = /
Process         PID      Real PID
Status          Container
init             1        1
sleep           root container
sh               2        2
sleep           root container

Container 2 : test_cont1 running ,
root path = /test_dir/test_cont1
Process         PID      Real PID
Status          Container
init             1        1
sleep           test_cont1
ps               2        8
running         test_cont1
```

5.6 pause the container

```
$ cont pause test_cont1
$ Container test_cont1 pause succeeds
$ ps

Container 1 : root container running ,
root path = /
Process         PID      Real PID
Status          Container
```

```
init                 1          1
sleep               root  container
sh                   2          2
sleep               root  container
ps                   12         12
running             root  container

Container 2 : test_cont1 paused  ,
root path = /test_dir/test_cont1
Process             PID        Real PID
Status              Container
init                 1          1
sleep               test_cont1
cont                 2          10
runnable            test_cont1
```

5.7 resume the container

```
$ cont resume test_cont1
Current working directory is
/test_dir/test_cont1
Root directory for container
test_cont1 is /test_dir/test_cont1
Container test_cont1 resume succeeds
ds
$ ps

Container 1 : root container running ,
root path = /
Process             PID        Real PID
Status              Container
init                 1          1
sleep               root  container
sh                   2          2
sleep               root  container
ps                   14         14
running             root  container

Container 2 : test_cont1 runnable,
root path = /test_dir/test_cont1
Process             PID        Real PID
Status              Container
init                 1          1
sleep               test_cont1
```

5.8 stop the container

```
$ cont stop test_cont1
Running container is rootcont
Container test_cont1 stop succeeds.
$ ps

Container 1 : root container running ,
root path = /
Process             PID        Real PID
Status              Container
init                 1          1
sleep               root  container
sh                   2          2
sleep               root  container
ps                   16         16
running             root  container
```

6 NEXT STEP

6.1 Known flaws

(1) standard stream(input, output and error stream) do not always function well. For example,

```
$ cd test_cont1
Ready to call sys_chdir...
Full path = /_cont1
Container's rootdir = /
cannot cd _cont1
$ (Note: its a newliner here)
$ test_
exec: fail
exec test_ failed
```

Here, the command is interpreted as two seperate commands. But this doesn't affect the overall functionality.

7 FEATURE TO PROSPECT

(1) Now the PID and file system isolation is completed as a whole, but for real container, say, Docker, it's allowed to isolated seperated by indicating distinct arguments.

(2) The normal way to exit processes is to call kill, but we met some unknown problems when implementing this idea. So we directly set its status to zombie, and wait for the initproc responsible for its deallocation and initialization. There could be more elegant way to do that.

8 LESSONS LEARNT

Working along this project, we read most of the xv6 source code, especially user programs, process and file system, which has close attachment to our projects. It's a great opportunity to get a close and in-depth understand of source code of a OS kernel, and we do learn a lot along the way, like the design of layers of file system – how to decouple every responsibility of file system, and allocate them to layers while providing interface to upper layer and other components.

9 CONCLUSION

Overall, we finished the goals set before. (1) implement several util commands for users to consume: we've added ps, pwd, and cont (2) complete PID isolation (3) complete file system isolation (4) do testing while developing