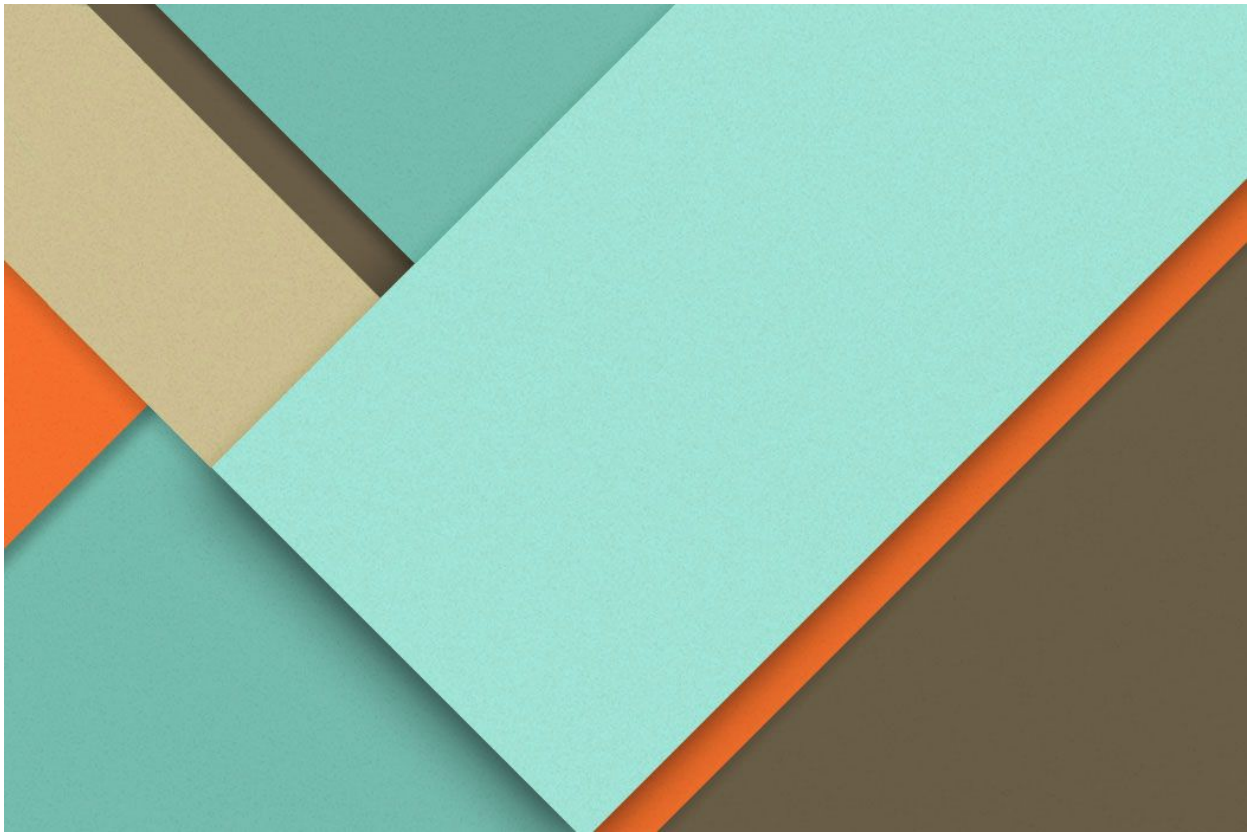

Assignment3

Operating System

26-04-2019

Aniket Kumar
2016CS50397

Sanket Hire
2016CS50402

IIT Delhi

Overview

In this assignment, we implemented container-based virtualization.

Some of the properties of container-based virtualization are:

a). OS acts as a base for container-engine.

b). Containers are used to provide virtualization.

c). Resources managed by base OS.

Container-based virtualization supports the following features :

a). Container - manager

b). Virtual Scheduler

c). System Calls

d). Virtual File System

Container Manager Implementation:

Data structure for container : Each container contains its unique container id, the last process it schedules. inum_arr, not_inum_arr ... are used in file system.

```
struct container {
  int cid;
  int num_proc;
  int last_proc;
  uint inum_arr[20];
  uint not_inum_arr[20];
  int inum_head;
  int not_inum_head;
};
```

Data Structure for container table: Container table contains a spinlock lock to provide mutual exclusion, an array which tells which container is active, and an array of containers.

We initialise each of ctable data structure to 0, and assign at time of creating container.

```
struct {
  struct spinlock lock;
  int iscontainer[NCONTAINER];
  struct container container[NCONTAINER];
} ctable;
```

Number of containers is capped by NCONTAINER(which is 16 in this case).

Function: Scheduling process within a container.

Implementation: In this system, we implement in such a way that OS first schedules container and subsequently container schedules its processes. To implement it, we edit the scheduler() code in proc.c. We first iterate over all the containers and for each container we iterate over processes to schedule it.

SYSTEM CALLS

Create_container(int container_id):

Function: this function is used to create a container with given container id.

Initialize elements of the container with 0.

cid of container is set as container_id.

acquire(lock) is called whenever we access ctable to avoid any kind of data race in ctable.

Correspnding entry of iscontainer is set to 1, indicating that container is created at this index.

```
int create_container(int container_id)
{
  struct container mycontainer = {0};
  mycontainer.cid = container_id;
  mycontainer.num_proc = 0;
  mycontainer.last_proc = -1;
  acquire(&ctable.lock);
  ctable.iscontainer[container_id] = 1;
  ctable.container[container_id] = mycontainer;
  ctable.container[container_id].inum_head = 0;
  ctable.container[container_id].not_inum_head = 0;
  for(int i=0;i<20;i++){
    ctable.container[container_id].inum_arr[i] = 0;
    ctable.container[container_id].not_inum_arr[i] = 0;
  }
  release(&ctable.lock);
  return 0;
}
```

Destroy_container(container_id):

```
int destroy_container(int container_id)
{
  struct proc *p;
  int pid;

  if(container_id == 0){
    cprintf("Container 0 can't be destroyed\n");
    return -1;
  }
```

This function is used to destroy container.

When it is called, all its processes are killed , clearly shown from the code.

Also, its corresponding iscontainer id is reset ot 0, saying that it has no container with given container_id.

Join Container

Function : Used to assign process to a container.

Implementation: Whenever a process calls join_container, then its corresponding cid is set to container_id. Number of processes in container is increased by 1.

```
int join_container(int container_id)
{
// check if container exists
  acquire(&ctable.lock);
  if(ctable.iscontainer[container_id] == 0){
    release(&ctable.lock);
    return -1;
  }
  release(&ctable.lock);

  struct proc *curproc = myproc();
  curproc->cid = container_id;
  acquire(&ctable.lock);
  ctable.container[container_id].num_proc ++;
  release(&ctable.lock);
  return 0;
}
```

Leave Container

Function: This system call is used to remove a process from a container and send it to the kernel.

Implementation: Whenever a process calls leave_container, its cid is set to 0, (cid = 0) means it belongs to the kernel. Also, the number of processes is decreased by1 in the container.

```
int leave_container(void)
{
  struct proc *curproc = myproc();
  int old_cont_id = curproc->cid;
  if(old_cont_id == 0){
    cprintf("Doesn't exist in any container\n");
    return -1;
  }
  curproc->cid = 0;
  acquire(&ctable.lock);
  ctable.container[old_cont_id].num_proc --;
  ctable.container[0].num_proc ++;
  release(&ctable.lock);
  return 0;
}
```

Virtual Scheduler

Implementation :

Virtual Scheduler is implemented in such a way that it iterates over containers and then in the container, it schedules the process. In this way, fairness is ensured across the containers. I have also maintained (stored) last process scheduled by container, so each time container starts to schedule from last process(excluding last process). In this way fairness is ensured within the container also. Scheduling starts from the process just after last process.

Some snippets of code:

```
for(cid = 0;cid<NCONTAINER;cid++)
{
  int last_proc;
  acquire(&ctable.lock);
  last_proc = ctable.container[cid].last_proc;
  release(&ctable.lock);

  acquire(&ptable.lock);
  // for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    int i,j;
    for(i=0;i<NPROC;i++){
      j = (i+last_proc+1)%NPROC;
      p = &ptable.proc[j];
    if(p->cid != cid)
      continue;

    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;

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

Process Isolation : PS

Function ps :

This function is used to print all the currently not UNUSED process of the container.

Implementation:

The process joined to a container is visible to processes of only that container. So, when we call ps(), then we print processes of that container only.

```
int
ps(void)
{

  struct proc *p;
  struct proc *curproc = myproc();
  int container_id = curproc->cid;
  acquire(&ptable.lock);

  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED && p->cid == container_id){
      cprintf("pid:%d ",p->pid);
      cprintf("name:%s\n",p->name);
    }
  }
  release(&ptable.lock);
  return 0;
}
```

File System Isolation: ls command

Function :

The files created by one container should not come in the output of ls command invoked by another container.

It is implemented in a user program (ls.c), not as a system call.

In this program, first print if the file is in that container and then print it otherwise don't.

I have implemented a function owner_right which takes input as cid and inum and tells whether that file belongs to that particular container cid.

If it belongs then print that file else don't.

```
int cid = get_cid();
switch(st.type){
case T_FILE:
  if(owner_right(cid,st.ino)){

  printf(1, "%s %d %d %d\n", fmtname(path), st.type, st.ino, st.size);
  }
  break;

case T_DIR:
  if(strlen(path) + 1 + DIRSIZ + 1 > sizeof buf){
    printf(1, "ls: path too long\n");
    break;
  }
  strcpy(buf, path);
  p = buf+strlen(buf);
  *p++ = '/';
  while(read(fd, &de, sizeof(de)) == sizeof(de)){
    if(de.inum == 0)
      continue;
    memmove(p, de.name, DIRSIZ);
    p[DIRSIZ] = 0;
    if(stat(buf, &st) < 0){
      printf(1, "ls: cannot stat %s\n", buf);
      continue;
    }

    if(owner_right(cid,st.ino)){
    printf(1, "%s %d %d %d\n", fmtname(buf), st.type, st.ino, st.size);
    }
  }
  break;
```

Copy On Write Mechanism

When opening a file which belongs to kernel (container 0), each file has permission to read and write that file. When any container write that file then that file is first copied to its container and then write takes place in that file. Whenever another container reads that file it doesn't read the modified file by former container.

This part is implemented in openfile (sysfile.c) .

```
uint inum_val = ip->inum;
if (owner_file(cid, inum_val) == 0)
  return -1;

if (((omode & O_WRONLY) || (omode & O_RDWR)) && cid != 0)
{
  // append string
  // create new path
  char *t1 = path;
  char *npath = (char *)kalloc();
  char *t2 = npath;
  while (*t1 != '\0')
  {
    *t2 = *t1;
    t2++;
    t1++;
  }
  *t2 = '-';
  t2++;
  *t2 = '0' + myproc()->cid;
  t2++;
  *t2 = '\0';
  struct inode *new_ip = create_file(npath);
  maintain_inum(myproc()->cid, new_ip->inum,ip->inum);
  // maintain_creations_helper( cid, new_ip->inum);
  char buf[512];
  int off = 0;
  int n = 0;
  if ((n = readi(ip, buf, off, sizeof(buf))) > 0)
  {
    writei(new_ip, buf, off, n);
    off += n;
```