# Assignment 3 OS

Ansh Prakash
2016CS10367

Himanshu
2016CS10348

IMPLEMENTATION

Each process is have a cid in its struct proc,which tells to which container it belongs.

Our Implementation is most in user space with some tweeks in the kernel space.

**Create_container**:
This initiliasizes the data structure required for the implementation

**Leave_container:**
This simply free the process from  the container by assigning it the cid =-1

**Join_container:**
This change the cwd of the process to container dir and provide it a slot in contiainer process table

**Destroy_Container:**
Kills all the process running in it.Empty the container folder(i.e delete all the files for the container)

*code in kernel space*

```
int
create_container(int cid){
  struct container container;
  if (no_of_containers == 0){
    // Initialising the Data Structure for the first time(assume we never delete a container(which is
not true))
    for (int i = 0; i < MAX_CONTAINER; ++i){
      struct container temp;
      temp.cid =-1;
      containers[i] = temp;
    }
  }
  if (no_of_containers+1 == MAX_CONTAINER){
    return(-1);
  }
  no_of_containers += 1;
  containers[cid] = container;
  container.cid =cid;
  container.current_running_process =-1;
  container.next_pid = 0;
  for(int i=0;i<MAX_NPROC;i++){
    struct proc p;
    p.pid = -1;
    container.ptable.proc[i] = p;
  }
  cprintf("Created Container %d \n",cid);
  return((uint)cid);
}
```

*DESTROY_CONTAINER*

```
uint
destroy_container ( uint container_id )
{
  containers[container_id].cid = -1;
  acquire(&helperlock.lock);
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->cid == container_id){
      kill(p->pid);
    }
  }
  release(&helperlock.lock);
  cprintf("Destroy Container %d \n",container_id);
  return(container_id);
}
```

JOIN_CONTAINER

```
uint
join_container(uint container_id)
{
  // struct cptable p = containers[container_id].ptable;
  struct proc *p = myproc();
  if(p->cid != -1 || containers[container_id].cid == -1){
    return(-1);
  }
  p->cid = container_id;
  // p->state = WAITING;
  if (containers[container_id].next_pid == MAX_NPROC-1){
    return(-1);
  }
  containers[container_id].ptable.proc[containers[container_id].next_pid] = *p;
  containers[container_id].next_pid +=1;
  cprintf("Joined Container %d ",container_id);
  cprintf("my pid is  %d \n",p->pid);
  return(0);
}
```

*Leave Container*

```
uint
leave_container(void)
{
  struct proc *p = myproc();
  int cid = p->cid;
  p->cid = -1;
  cprintf("Leave Container %d \n",cid);
  return(0);
}
```

PS:
This is Implemented in Kernel space

```
void
ps()
{
  struct proc *mp = myproc();
  int mycid = mp->cid;
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED && p->cid == mycid){
      cprintf("pid:%d name:%s\n",p->pid,p->name);
    }
  }
  release(&ptable.lock);
}
```

Helper sys call to get cid in user space

```
int
getcid(){
  struct proc *mp = myproc();
  int mycid = mp->cid;
  return(mycid);
}
```

*PROC STRUCT:*

```
// Per-process state
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 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)
  int cid;
};
```

CONTAINER STRUCT

```
struct container {
    int cid;
    int next_pid;
    int current_running_process;
    struct cptable ptable;
};
```

USER SPACE
CODE
ls and COW-mechanism have been implemented in USER space.

```
#include "types.h"
#include "stat.h"
#include "user.h"
#include "fs.h"
#include "fcntl.h"

//convert int to string
void
itoa(int num , char* ret)
{
  int rem, len = 0;
  int n = num;
  while(n!=0){
    len++;
    n = n/10;
  }
  for(int i=0;i<len;i++){
    rem = num%10;
    num = num/10;
    ret[len - (i+1)] =rem +'0';
  }
  ret[len] = '\0';
}


char*
my_fmtname(char *path)
{
  static char buf[DIRSIZ+1];
  char *p;

  // Find first character after last slash.
  for(p=path+strlen(path); p >= path && *p != '/'; p--)
    ;
  p++;

  // Return blank-padded name.
  if(strlen(p) >= DIRSIZ)
    return p;
  memmove(buf, p, strlen(p));
  memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));
  return buf;
}

void
my_ls(char *path)
{
  char buf[512], *p;
  int fd;
  struct dirent de;
  struct stat st;

  if((fd = open(path, 0)) < 0){
    printf(2, "ls: cannot open %s\n", path);
    return;
  }

  if(fstat(fd, &st) < 0){
    printf(2, "ls: cannot stat %s\n", path);
    close(fd);
    return;
  }

  switch(st.type){
  case T_FILE:
    printf(1, "%s\n", my_fmtname(path));
    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;
      }
      printf(1, "%s\n", my_fmtname(buf));
    }
    break;
  }
  close(fd);
}

void
destroy_container_wrapper(int cid)
{
  char path[4];
  path[0] = 'c';
  char  conv[3];
  itoa(cid,conv);
  path[1] = conv[0];
  path[2] = conv[1];
  path[3] = '\0';


  char deldir[7];
  deldir[0] = '.';
  deldir[1] = '.';
  deldir[2] = '/';
  for(int i=0 ;i<4;i++){
    deldir[3+i] = path[i];
  }

  unlink(deldir);
  mkdir(path);
  destroy_container(cid);
}

void
cont_ls()
{
  my_ls(".");
}

void
init_containers_resources(){
  //setting up the file system
  mkdir("c1");
  mkdir("c2");
  mkdir("c3");
  mkdir("c4");
  mkdir("c5");
  mkdir("c6");
  mkdir("c7");
  mkdir("c8");
  mkdir("c9");
  mkdir("c10");
  mkdir("c11");
  mkdir("c12");
  mkdir("c13");
  mkdir("c14");
  mkdir("c15");
  mkdir("c16");
  mkdir("c17");
  mkdir("c18");
  mkdir("c19");
  mkdir("c20");
}



void
join_container_wrapper(int i){
  char cid_s[4];
  cid_s[0] = 'c';
  char  conv[3];
  itoa(i,conv);
  cid_s[1] = conv[0];
  cid_s[2] = conv[1];
  cid_s[3] = '\0';
  chdir(cid_s);
  join_container(i);
}
```

```
int
leave_container_wrapper()
{
  chdir("/");
  return(leave_container());
}

void
create_file(char* filename)
{
  int fd;
  if((fd = open(filename,O_CREATE))<0){
    printf(1," not able to create the file\n");
  }
  close(fd);
}


void
copyfile(char* file1,char* file2)
{
  int fd0,fd1,n;
  char buf[512];
  if((fd0 = open(file1,O_RDONLY))<0){
    printf(1," error while copying \n");
  }
  create_file(file2);
  if((fd1 = open(file1,O_RDWR))<0){
    printf(1," error while copying \n");
  }
  while((n = read(fd0,buf,sizeof(buf))) >0 ){
    write(fd1,buf,n);
  }
  close(fd0);
  close(fd1);
}

/*
 *   mode : rdonly 0
 *        : wronly 1
 *        : rdwr 2
 */
int
open_file(char* filename,int mode)
{
  int fd = -1;
  if(mode == 0 ){
    fd = open(filename,O_RDONLY);
  }
  else if(mode == 1){
    fd = open(filename,O_WRONLY);
  }
  else if(mode == 2){
    fd = open(filename,O_RDWR);
  }

  if(fd<0){
      char path[30];
      path[0] = '/';
      int i = 0;
      while(filename[i]!='\0'){
        path[i+1] = filename[i];
        i++;
      }
      if((fd = open(path,O_RDWR))<0){
        printf(1," no such file exist\n");
        return(-1);
      }
      int cid = getcid();
      char path_t[40];
      path_t[0] = '/';
      path_t[1] ='c';
      char  conv[3];
      itoa(cid,conv);
      int j = 2;
      while(conv[j-2]!='\0'){
        path_t[j] = conv[j-2];
        printf(0,"%s\n", conv);
        j++;
      }
      path_t[j] = '/';
      j++;
      i = 0;
      while(filename[i]!='\0'){
        path_t[i+j] = filename[i];
        i++;
      }
      path_t[i+j] = '\0';
      if(mode == 0 ){
        fd = open(path,O_RDONLY);
      }
      else if(mode == 1){
        if((fd = open(path,O_WRONLY))<0){
          printf(1," no such file exist\n");
        }
        close(fd);
        copyfile(path,path_t);
        fd = open(filename,O_WRONLY);
        return(fd);
      }
      else if(mode == 2){
        if((fd = open(path,O_RDWR))<0){
          printf(1," no such file exist\n");
        }
        close(fd);
        printf(0,"file apth_t %s\n",path_t);
        copyfile(path,path_t);

        fd = open(filename,O_RDWR);
        return(fd);
      }
      if(fd<0){
        printf(1," no such file exist\n");
      }
  }
  return(fd);
}


void
cont_cat(char* file)
{
  int fd,n;
  char buf[512];
  if((fd = open_file(file,0))<0){
    printf(1," error while copying \n");
  }

  while((n = read(fd,buf,sizeof(buf))) >0 ){
    printf(0,"%s",buf);
  }
  close(fd);
}

int main(void)
{

  init_containers_resources();
              create_container(1);
  create_container(2);
  create_container(3);
  //////////////FILE ISOLATION//////////////////////
  int pid = fork();
  if(pid == 0){
    join_container_wrapper(1); // called only by child created by preceeding fork call .
    create_file("hello");
    int fd;
    fd =open_file("hello",2);
    char raw[5] = "ansh\n";
    int n =5;
    write(fd,raw,n);
    close(fd);
    printf(0,"Testing ls \n\n");
    cont_ls();
    // printf(0,"Testing ps \n\n\n");
    // ps();
    exit();
  }
  // destroy_container_wrapper(1);

  pid = fork();
  if(pid == 0){
    join_container_wrapper(2); // called only by child created by preceeding fork call .
    create_file("YOYO");
    int fd;
    fd =open_file("YOYO",2);
    char raw[5] = "five\n";
    int n =5;
    write(fd,raw,n);
    close(fd);
    printf(0,"Testing ls \n\n\n");
    cont_ls();
    printf(0,"Testing ps \n\n\n");
    ps();
    exit();
  }
  ////////FILE ISOLATION ENDS//////

  /////////////COPY ON WRITE TESTING //////
  create_file("ROOT");
  int fdt;
  fdt =open_file("ROOT",2);
  char raw[5] = "root\n";
  int n =5;
  write(fdt,raw,n);
  close(fdt);

  int pid = fork();
  if(pid == 0){
    join_container_wrapper(2); // called only by child created by preceeding fork call .
    int fd;
    cont_cat("ROOT");//This will read from the root file
    fd =open_file("ROOT",2);
    char raw[9] = "anshroot\n";
    int n =9;
    write(fd,raw,n);
    close(fd);
    printf(0,"Testing ls \n\n");
    cont_ls();
    cont_cat("ROOT");
    exit();
  }
          /////////////COPY ON WRITE TESTING //////
          wait();
  exit();
}
```

*FILE ISOLATION STRATEDY:*

*For resource isolation the user spcae(create_container.c) code first create folders for each of the possible container and then any file created in that kernel will be stored in that folder.*
*This implementation keep in mind that the container process can see the root dir.*

***Special functionality***
*void copyfile(char* file1,char* file2);*
*This function copy the content of file1 to file 2.*

---