컴퓨터학부 20132397 윤건택

1. 과제 개요

이번 과제는 파일 시스템을 코딩함으로써 파일 시스템들이 어떻게 동작하는지에 관한 과제다. SSUOS의 파일 시스템은 procfs와 ssufs로 이뤄져 있는데, 내부의 inode와 vnode의 상관관계를 이해하고 트리를 구축하며, ssufs와 procfs의 차이점을 인식하고 그 파일 시스템을 다루는 함수를 코딩하여 시스템에 관한 이해도를 높일 수 있도록 한다.

1. 설계
2. make\_vnode\_tree() 구현

vnode 트리를 구현하기 위해선 우선 ssufs\_inode의 root inode로 부터 시작해야한다. 루트 vnode의 parent node는 자기 자신이 되므로 set\_vnode를 이용해 parent node를 할당함과 동시에 ssufs\_inode\_table에 있는 root\_inode를 mnt\_node의 info에 할당한다. 그 이후 ssufs\_inode\_read 함수를 사용하여 vnode의 info가 가리키는 inode의 i\_direct구조체를 필요한 길이만큼 잘라서 자식 inode를 불러온 후, set\_vnode함수를 이용해 자식노드들에게 부모노드를 지정해주고 이 함수를 재귀적으로 불러서 트리를 작성한다.

1. ssufs\_mkdir() 구현

현재 프로세스의 vnode에 directory 형식의 자식 vnode를 처리하는 것으로 우선, inode를 생성해 이 inode에 타입에 디렉토리 타입을 명시하고 “.”, “..” 디렉토리를 ssufs\_inode\_write를 이용해 넣어준다. 그 이후 임시 vnode를 생성해 필요한 값들을 명시해주고 이후 set\_vnode 함수를 통해 부모노드와의 연결 및 inode와의 연결을 시행한다.

1. procfs 구현

procfs에 할당된 vnode는 용도별로 vnode.v\_op의 값들이 다르므로 유의해야한다. 처음 proc\_process\_cd는 pid를 검색해 입력 받은 값과 pid가 동일 할 경우 vnode를 생성하고 내부에 v\_op를 지정해준 후 부모 프로세스와 연결을 시행한다. proc\_process\_info\_cd 함수는 프로세스의 속성에 접근하는 함수인데, 현재 pid값을 알 수 있는 것은 그전에 지정해놓은 vname이므로 이 값과 pid를 비교하여 해당 프로세스를 지정하고, 입력 받은 cwd와 root에 따라 vnode를 process의 어떤 vnode를 복사할지를 정한다. 그 이후 부모노드를 지정해준다. proc\_link\_ls 함수는 현재 vnodㄹ의 하위 디렉토리를 출력하는 것과 동일하므로 childlist에 있는 노드의 이름을 출력해주는 것으로 구현한다. proc\_process\_info\_cat은 우선 vname과 pid를 비교해 값을 읽어와야 할 프로세스를 지정 한 후 입력 받은 값이 stack인지 time인지에 따라 필요한 정보를 출력한다.

1. 실행 결과



<그림 1>



<그림 2>



<그림 3>

1. 소스 코드

ssufs.c

#include <filesys/ssufs.h>

#include <filesys/vnode.h>

#include <device/block.h>

#include <device/ata.h>

#include <proc/proc.h>

#include <ssulib.h>

#include <string.h>

#include <bitmap.h>

#define MIN(a, b) (a<b?a:b)

static unsigned char bitmapblock[SSU\_BLOCK\_SIZE];

static struct ssufs\_superblock ssufs\_sb;

extern struct blk\_dev ata1\_blk\_dev;

extern struct process \*cur\_process;

char tmpblock[SSU\_BLOCK\_SIZE];

struct vnode \*init\_ssufs(char \*volname, uint32\_t lba, struct vnode \*mnt\_root){

int result;

int i;

char superblock[SSU\_BLOCK\_SIZE];

ssufs\_sb.blkdev = &ata1\_blk\_dev;

result = ssufs\_readblock(&ssufs\_sb, lba, superblock);

memcpy(&ssufs\_sb, superblock, sizeof(ssufs\_sb));

ssufs\_sb.blkdev = &ata1\_blk\_dev;

if(ssufs\_sb.sb\_magic != SSU\_SB\_MAGIC){

ssufs\_sb.sb\_nblocks = (ssufs\_sb.blkdev->blk\_count / (SSU\_BLOCK\_SIZE / ssufs\_sb.blkdev->blk\_size));

ssufs\_sb.lba = lba;

// sb.sb->ssufs\_info->sb\_nblocks;

}else{

}

//load or init bitmap block

ssufs\_load\_databitmapblock(&ssufs\_sb);

ssufs\_load\_inodebitmapblock(&ssufs\_sb);

ssufs\_sync(&ssufs\_sb);

//laod or init inode table

ssufs\_load\_inodetable(&ssufs\_sb);

return make\_vnode\_tree(&ssufs\_sb, mnt\_root);

}

int ssufs\_load\_inodetable(struct ssufs\_superblock \*sb)

{

int result = 0;

int i;

struct ssufs\_inode \*root\_inode;

struct dirent dirent;

for(i=0; i<NUM\_INODE\_BLOCK; i++)

ssufs\_readblock(sb, SSU\_INODE\_BLOCK(sb->lba) + i, ((char\*)ssufs\_inode\_table) + (i \* SSU\_BLOCK\_SIZE));

//no root directory

if(!bitmap\_test(sb->inodemap, INODE\_ROOT)){

memset(ssufs\_inode\_table, 0x00, sizeof(struct ssufs\_inode) \* NUM\_INODE);

//Unvalid, Reserved

bitmap\_set(sb->inodemap, 0, true);

bitmap\_set(sb->inodemap, 1, true);

bitmap\_set(sb->inodemap, INODE\_ROOT, true);

root\_inode = &ssufs\_inode\_table[INODE\_ROOT];

root\_inode->i\_no = INODE\_ROOT;

root\_inode->i\_size = 0;

root\_inode->i\_type = SSU\_DIR\_TYPE;

root\_inode->i\_refcount = 1;

root\_inode->ssufs\_sb = sb;

//set root dirent

dirent.d\_ino = INODE\_ROOT;

dirent.d\_type = SSU\_DIR\_TYPE;

memcpy(dirent.d\_name, ".", sizeof("."));

ssufs\_inode\_write(root\_inode, 0, (char \*)&dirent, sizeof(struct dirent));

memcpy(dirent.d\_name, "..", sizeof(".."));

ssufs\_inode\_write(root\_inode, root\_inode->i\_size, (char \*)&dirent, sizeof(struct dirent));

ssufs\_sync\_bitmapblock(sb);

ssufs\_sync\_inodetable(sb);

printk("initialize inodetable : %d\n", NUM\_INODE);

}else{

printk("load inodetable : %d\n", NUM\_INODE);

}

return result;

}

int ssufs\_load\_databitmapblock(struct ssufs\_superblock \*sb)

{

int result;

int i;

//load bitmap block from HDD

result = ssufs\_readblock(sb, SSU\_BITMAP\_BLOCK(sb->lba), bitmapblock);

//load block bitmap

sb->blkmap = (struct bitmap\*)bitmapblock;

if(sb->blkmap->bit\_cnt != sb->sb\_nblocks){

sb->blkmap = bitmap\_create\_in\_buf(ssufs\_sb.sb\_nblocks, bitmapblock, SSU\_BLOCK\_SIZE/2);

for(i=0; i<SSU\_BITMAP\_BLOCK(sb->lba); i++)

{

bitmap\_set(sb->blkmap, i, true);

}

printk("initialize block bitmap : %d bit\n", sb->blkmap->bit\_cnt);

}else{

printk("load block bitmap : %d bit\n", sb->blkmap->bit\_cnt);

}

return result;

}

int ssufs\_load\_inodebitmapblock(struct ssufs\_superblock \*sb)

{

//load inode bitmap

sb->inodemap = (struct bitmap\*)(bitmapblock + SSU\_BLOCK\_SIZE/2);

if(sb->inodemap->bit\_cnt != NUM\_INODE)

{

sb->inodemap = bitmap\_create\_in\_buf(NUM\_INODE, bitmapblock + SSU\_BLOCK\_SIZE/2, SSU\_BLOCK\_SIZE/2);

printk("initialize inode bitmap : %d bit\n", sb->inodemap->bit\_cnt);

}

else

printk("load inode bitmap : %d bit\n", sb->inodemap->bit\_cnt);

}

int ssufs\_readblock(struct ssufs\_superblock \*sb, uint32\_t blknum, char \*buf)

{

int result = 0;

int startsec = blknum \* SECTORCOUNT(sb->blkdev);

int i;

for(i=0; i < SECTORCOUNT(sb->blkdev); i++){

DEVOP\_READ(sb->blkdev, startsec + i, buf + (i \* sb->blkdev->blk\_size));

}

return result;

}

int ssufs\_writeblock(struct ssufs\_superblock \*sb, uint32\_t blknum, char \*buf)

{

int result = 0;

int startsec = blknum \* SECTORCOUNT(sb->blkdev);

int i;

for(i=0; i<SECTORCOUNT(sb->blkdev); i++)

DEVOP\_WRITE(sb->blkdev, startsec + i, buf + (i \* sb->blkdev->blk\_size));

return result;

}

int ssufs\_sync\_superblock(struct ssufs\_superblock \*sb){

int result = 0;

ssufs\_writeblock(sb, sb->lba, (char \*)sb);

return result;

}

int ssufs\_sync\_bitmapblock(struct ssufs\_superblock \*sb){

int result = 0;

ssufs\_writeblock(sb, SSU\_BITMAP\_BLOCK(sb->lba), sb->blkmap);

return result;

}

int ssufs\_sync\_inodetable(struct ssufs\_superblock \*sb){

int result = 0;

int i;

for(i = 0 ; i < NUM\_INODE\_BLOCK; i++)

ssufs\_writeblock(sb, SSU\_INODE\_BLOCK(sb->lba)+i, ((char \*)ssufs\_inode\_table+(i\*SSU\_BLOCK\_SIZE)));

return result;

}

int ssufs\_sync(struct ssufs\_superblock \*sb){

int result = 0;

ssufs\_sync\_superblock(sb);

ssufs\_sync\_bitmapblock(sb);

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* inode start \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int ssufs\_inode\_write(struct ssufs\_inode \*inode, uint32\_t offset, char \*buf, uint32\_t len){

int result = 0;

uint32\_t blkoff;

uint32\_t resoff;

int block\_index;

int i;

if(offset > inode->i\_size || len <= 0 || buf == NULL)

return -1;

for(i = offset; i < offset+len; i = ((blkoff+1)\*SSU\_BLOCK\_SIZE))

{

blkoff = i / SSU\_BLOCK\_SIZE;

resoff = i % SSU\_BLOCK\_SIZE;

memset(tmpblock, 0, SSU\_BLOCK\_SIZE);

if(blkoff < NUM\_DIRECT){//direct

if(inode->i\_direct[blkoff] == 0){//need to bitmap\_alloc

block\_index = bitmap\_alloc(inode->ssufs\_sb->blkmap);

block\_index += SSU\_DATA\_BLOCK(inode->ssufs\_sb->lba);

if(block\_index == -1){

return -1;

}

inode->i\_direct[blkoff] = block\_index;

}else{

ssufs\_readblock(inode->ssufs\_sb, inode->i\_direct[blkoff], tmpblock);

struct dirent dirent;

memcpy(&dirent, tmpblock, sizeof(dirent));

}

memcpy(tmpblock+resoff, buf, len);

ssufs\_writeblock(inode->ssufs\_sb, inode->i\_direct[blkoff], tmpblock);

}else{//indirect,. not used

}

}

inode->i\_size = offset+len;

ssufs\_sync\_bitmapblock(inode->ssufs\_sb);

ssufs\_sync\_inodetable(inode->ssufs\_sb);

return result;

}

int ssufs\_inode\_read(struct ssufs\_inode \*inode, uint32\_t offset, char \*buf, uint32\_t len)

{

int result = 0;

uint32\_t blkoff;

uint32\_t resoff;

int i;

if(offset > inode->i\_size)

return -1;

for(i=0; i < len; i+=(SSU\_BLOCK\_SIZE - resoff)){

blkoff = offset / SSU\_BLOCK\_SIZE;

resoff = offset % SSU\_BLOCK\_SIZE;

if(blkoff < NUM\_DIRECT){//direct

ssufs\_readblock(inode->ssufs\_sb, inode->i\_direct[blkoff], tmpblock);

memcpy(buf, tmpblock+resoff, MIN(SSU\_BLOCK\_SIZE - resoff, len));

}else{//indirect, #not used

}

}

return result;

}

struct ssufs\_inode \*inode\_alloc(uint32\_t type){

struct ssufs\_inode \*cwd\_inode = (struct ssufs\_inode\*)(cur\_process->cwd->info);

struct ssufs\_superblock \*sb = cwd\_inode->ssufs\_sb;

struct ssufs\_inode \*inode;

int i;

for(i = INODE\_ROOT; i < NUM\_INODE; i++){

if(!bitmap\_test(sb->inodemap, i)){

inode = &ssufs\_inode\_table[i];

inode->i\_no = i;

inode->i\_size = 0;

inode->i\_type = type;

inode->i\_refcount = 1;

inode->ssufs\_sb = sb;

bitmap\_set(sb->inodemap, i, true);

ssufs\_sync\_inodetable(sb);

ssufs\_sync\_bitmapblock(sb);

return inode;

}

}

return NULL;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* inode end \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

struct vnode \*make\_vnode\_tree(struct ssufs\_superblock \*sb, struct vnode \*mnt\_root)

{

struct ssufs\_inode \*iret;

struct vnode \*vret;

struct dirent buf;

int num\_dir;

if(sb != NULL)

set\_vnode(mnt\_root, mnt\_root, &ssufs\_inode\_table[2]);

num\_dir = num\_direntry((struct ssufs\_inode \*)mnt\_root->info);

for(int i = 2; i < num\_dir; i++){

ssufs\_inode\_read((struct ssufs\_inode \*)mnt\_root->info,i \* sizeof(struct dirent), (char\*) &buf, sizeof(struct dirent));

vret = vnode\_alloc();

memset(vret->v\_name, 0,LEN\_VNODE\_NAME);

set\_vnode(vret, mnt\_root,&ssufs\_inode\_table[buf.d\_ino]);

make\_vnode\_tree(NULL, vret);

list\_push\_back(&mnt\_root->childlist, &vret->elem);

}

return mnt\_root;

}

int num\_direntry(struct ssufs\_inode \*inode)

{

if(inode->i\_size % sizeof(struct dirent) != 0 || inode->i\_type != SSU\_DIR\_TYPE)

return -1;

return inode->i\_size / sizeof(struct dirent);

}

void set\_vnode(struct vnode \*vnode, struct vnode \*parent\_vnode, struct ssufs\_inode \*inode)

{

struct dirent dirent;

struct ssufs\_inode \*parent\_inode;

int i, ndir;

get\_curde(inode, &dirent);

memcpy(vnode->v\_name, dirent.d\_name, LEN\_VNODE\_NAME);

vnode->v\_parent = parent\_vnode;

vnode->type = inode->i\_type;

vnode->v\_op.mkdir = ssufs\_mkdir;

vnode->v\_op.ls = NULL;

vnode->info = (void \*)inode;

}

int get\_curde(struct ssufs\_inode \*cwd, struct dirent \* de)

{

struct ssufs\_inode \*pwd;

int i, ndir;

//get parent dir

ssufs\_inode\_read(cwd, sizeof(struct dirent), (char\*)de, sizeof(struct dirent));

pwd = &ssufs\_inode\_table[de->d\_ino];

ndir = num\_direntry(pwd);

for(i=0; i<ndir; i++)

{

ssufs\_inode\_read(pwd, i\*sizeof(struct dirent), (char\*)de, sizeof(struct dirent));

if(de->d\_ino == cwd->i\_no)

return 0;

}

return -1;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* vnode operation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int ssufs\_mkdir(char \*dirname){

struct vnode \*vret;

struct ssufs\_inode \*iret;

struct dirent dirent;

//inode 생성 및 초기화

if((iret = inode\_alloc(SSU\_DIR\_TYPE)) == NULL)

printk("inode\_alloc failed!\n");

//생성될 inode에 대한 dirent 구조체 입력

dirent.d\_ino = 0;

dirent.d\_type = SSU\_DIR\_TYPE;

memset(dirent.d\_name, 0, FILENAME\_LEN);

memcpy(dirent.d\_name, ".", sizeof("."));

ssufs\_inode\_write(iret, 0, (char \*)&dirent, sizeof(struct dirent));

dirent.d\_ino = ((struct ssufs\_inode\*)cur\_process->cwd->info)->i\_no;

memset(dirent.d\_name, 0, FILENAME\_LEN);

memcpy(dirent.d\_name,"..", sizeof(".."));

ssufs\_inode\_write(iret,iret->i\_size, (char \*)&dirent, sizeof(struct dirent));

//vnode 생성 및 초기화

if((vret = vnode\_alloc()) == NULL)

printk("vnode\_alloc failed!\n");

vret->type = iret->i\_type;

vret->info = (void \*)iret;

set\_vnode(vret, cur\_process->cwd, iret);

memset(vret->v\_name, 0, LEN\_VNODE\_NAME);

memcpy(vret->v\_name, dirname, strlen(dirname));

dirent.d\_ino = iret->i\_no;

memset(dirent.d\_name, 0, FILENAME\_LEN);

memcpy(dirent.d\_name, dirname, strlen(dirname));

list\_push\_back(&cur\_process->cwd->childlist, &vret->elem);

ssufs\_inode\_write((struct ssufs\_inode\*)cur\_process->cwd->info, ((struct ssufs\_inode\*)cur\_process->cwd->info)->i\_size, (char \*)&dirent, sizeof(struct dirent));

return 0;

}

procfs.c

#include <filesys/procfs.h>

#include <filesys/vnode.h>

#include <filesys/ssufs.h>

#include <proc/proc.h>

#include <list.h>

#include <string.h>

#include <ssulib.h>

extern struct list p\_list;

extern struct process \*cur\_process;

struct vnode \*init\_procfs(struct vnode \*mnt\_vnode)

{

mnt\_vnode->v\_op.ls = proc\_process\_ls;

mnt\_vnode->v\_op.mkdir = NULL;

mnt\_vnode->v\_op.cd = proc\_process\_cd;

return mnt\_vnode;

}

int proc\_process\_ls()

{

int result = 0;

struct list\_elem \*e;

printk(". .. ");

for(e = list\_begin (&p\_list); e != list\_end (&p\_list); e = list\_next (e))

{

struct process\* p = list\_entry(e, struct process, elem\_all);

printk("%d ", p->pid);

}

printk("\n");

return result;

}

int proc\_process\_cd(char \*dirname)

{

struct list\_elem \*e;

struct vnode \*cwd = cur\_process->cwd;

struct process \* p;

struct vnode \* vret;

char input[64];

for(e = list\_begin(&p\_list); e != list\_end(&p\_list);e = list\_next(e))

{

p = list\_entry(e, struct process, elem\_all);

snprintf(input,sizeof(p->pid), "%d", p->pid);

if(strcmp(input, dirname) == 0){

vret = vnode\_alloc();

vret->v\_parent = cwd;

memset(vret->v\_name, 0, LEN\_VNODE\_NAME);

memcpy(vret->v\_name, input, strlen(input));

vret->v\_op.ls = proc\_process\_info\_ls;

vret->v\_op.cd = proc\_process\_info\_cd;

vret->v\_op.cat = proc\_process\_info\_cat;

cur\_process->cwd = vret;

return 0;

}

}

return 0;

}

int proc\_process\_info\_ls()

{

int result = 0;

printk(". .. ");

printk("cwd root time stack");

printk("\n");

return result;

}

int proc\_process\_info\_cd(char \*dirname)

{

struct list\_elem \*e;

struct vnode \*cwd = cur\_process->cwd;

struct process \*p;

struct vnode \* vret;

int input[64];

for(e = list\_begin(&p\_list); e != list\_end(&p\_list); e = list\_next(e))

{

p = list\_entry(e, struct process, elem\_all);

snprintf(input, sizeof(p->pid), "%d", p->pid);

if(strcmp(cwd->v\_name, input) == 0){

if(strcmp(dirname, "cwd") == 0){

vret = vnode\_alloc();

copy\_vnode(p->cwd, vret);

memset(vret->v\_name, 0, LEN\_VNODE\_NAME);

memcpy(vret->v\_name, "cwd", strlen("cwd"));

vret->v\_parent = cwd;

vret->v\_op.ls = proc\_link\_ls;

cur\_process->cwd = vret;

return 0;

}

if(strcmp(dirname, "root") == 0){

vret = vnode\_alloc();

copy\_vnode(p->rootdir, vret);

memset(vret->v\_name, 0, LEN\_VNODE\_NAME);

memcpy(vret->v\_name, "root", strlen("root"));

vret->v\_parent = cwd;

vret->v\_op.ls = proc\_link\_ls;

cur\_process->cwd = vret;

return 0;

}

}

}

return 0;

}

int proc\_process\_info\_cat(char \*filename)

{

struct list\_elem \*e;

struct vnode \*cwd = cur\_process->cwd;

struct process \*p;

int input[64];

for(e = list\_begin(&p\_list); e != list\_end(&p\_list); e = list\_next(e))

{

p = list\_entry(e, struct process, elem\_all);

snprintf(input, sizeof(p->pid), "%d", p->pid);

if(strcmp(cwd->v\_name, input) == 0){

if(strcmp(filename, "stack") == 0){

printk("stack : %x\n", p->stack);

}

if(strcmp(filename, "time") == 0){

printk("time : %ld\n", p->time\_used);

}

}

}

return 0;

}

int proc\_link\_ls()

{

struct list\_elem \*e;

struct vnode \*cwd =cur\_process->cwd;

struct vnode \*child;

int num\_dir = num\_direntry((struct ssufs\_inode \*) cwd ->info);

printk(". .. ");

e = list\_begin(&cwd->childlist);

for(int i = 2; i < num\_dir; i++)

{

child = list\_entry(e, struct vnode, elem);

printk("%s ", child->v\_name);

e = list\_next(e);

}

printk("\n");

}