Project 4: cpmFS - A Simple File System

Mousumi Akter

April 2020

1 Project Overview

The goal of this project is to design and implement a simple file system called cpmFS (i.e., CP/M file sytem). This simple file system allows users to list directory entries, rename files, copy files, delete files, as well as code to read-/write/open/close files.

2 Data Flow Diagrams

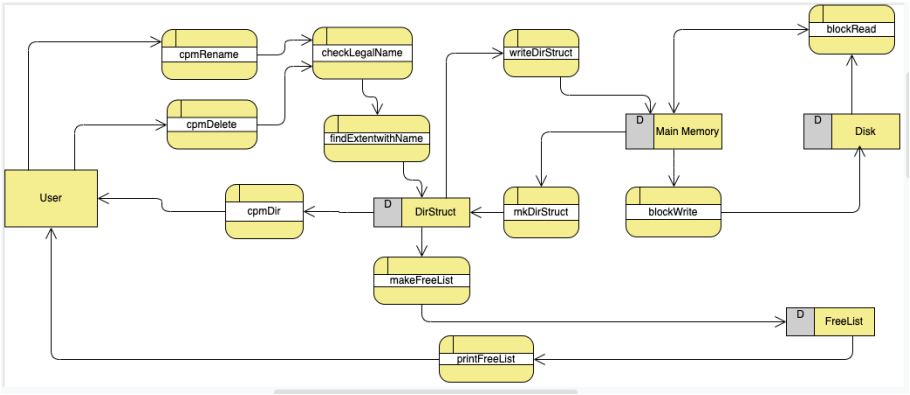

Figure 1: Data Flow Diagram of cpmFS

3 Algorithm Designs

3.1 mkDirStruct

This function allocates memory for a DirStructType and populates it, given a pointer to a buffer of memory holding the contents of disk block 0 (e), and an integer index, which tells which extent from block zero (extent numbers start with 0) to use to make the DirStructType value to return.

```
DirStructType *mkDirStruct(int index,uint8_t *e)
{
//create memory space for dir
        DirStructType *d;
        d = malloc(sizeof(DirStructType));

//obtain status from in-memory Block0
        d->status = (e+index*EXTENT_SIZE)[0];

//obtain name from in-memory Block0
//If name length <8, padded by ' '
        int i=1;
        char ch;
        for(; i<9; i++)
        {
                ch = (e+index*EXTENT_SIZE)[i];
                (d->name)[i-1] = ch;
                if(ch == ' ') break;
        }
        if(i<9 && ch ==' ')
        {
                d->name[i-1] = '\0';
        }
        if(i==9)
        {
                (d->name)[i] = '\0';
        }

//obtain dir->extension from in-memory Block0
        int extCount=0;
        i=9;
        for( ; i<12; i++)
        {
                ch = (e+index*EXTENT_SIZE)[i];
                (d->extension)[i-9] = ch;
                extCount++;
                if(ch==' ') break;

        }
        if(extCount<3 && ch == ' ')
        {
                (d->extension)[i-9] = '\0';
        }
        else
        {
```

```
                (d->extension)[i-8] = '\0';
        }

//obtain XL,BC,XH,RC from from in-memory Block0
        d->XL = (e+index*EXTENT_SIZE)[12];
        d->BC = (e+index*EXTENT_SIZE)[13];
        d->XH = (e+index*EXTENT_SIZE)[14];
        d->RC = (e+index*EXTENT_SIZE)[15];

//obtain all 16 fileblocks from in-memory Block0
        memcpy(d->blocks, e+index*EXTENT_SIZE+FILE_BLOCK_SIZE , FILE_BLOCK_SIZE)

//return dir
        return d;
}
```

3.2 writeDirStruct

This function writes contents of a DirStructType struct back to the specified index of the extent in block of memory (disk block 0) pointed to by e

```
void writeDirStruct(DirStructType *d, uint8_t index, uint8_t *e)
{

//write status from DirStruct to in-memory Block0
        (e+index*EXTENT_SIZE)[0] = d->status;

        int i;
        int extCount;


//write name from DirStruct to in-memory Block0
//If name length <8, pad with ' '
        i = 1;
        for(;d->name[i-1] != '\0';i++)
        {
            (e+index*EXTENT_SIZE)[i] = d->name[i-1];

            if(d->name[i-1] == '.') break;
        }
        if(i<9)
        {
                for(;i<9;i++)
                {
                    (e+index*EXTENT_SIZE)[i] = '␣';
                }
```

```
        }


//write extension from DirStruct to in-memory Block0
    extCount= 0;
    while(d->extension[extCount] != '\0')
    {
        (e+index*EXTENT_SIZE)[i] = d->extension[extCount];
        i++;
        extCount++;
        }

        if(i<12)
        {
                for(;i<12;i++)
            {
                (e+index*EXTENT_SIZE)[i] = '␣';
                }

        }
//write XL,BC,XH,RC from DirStruct to in-memory Block0
        (e+index*EXTENT_SIZE)[12] = d->XL;
        (e+index*EXTENT_SIZE)[13] = d->BC;
        (e+index*EXTENT_SIZE)[14] = d->XH;
        (e+index*EXTENT_SIZE)[15] = d->RC;

//write all 16 file blocks from DirStruct to in-memory Block0
        memcpy(e+index*EXTENT_SIZE+FILE_BLOCK_SIZE, d->blocks, FILE_BLOCK_SIZE);
}
```

3.3 makeFreeList

This function populates the FreeList global data structure. freeList[i] == true means that block i of the disk is free. block zero is never free, since it holds the directory. freeList[i] == false means the block is in use.

```
void makeFreeList()
{
        uint8_t block0[BLOCK_SIZE];

//initially set all blocks as free
        for (int i = 0; i < NUM_BLOCKS; i++)
        {
                freeList[i]=true;
```

```
        }

//set blocko as occupied
        freeList[0]=false;

//load block0 to main memory
        blockRead(block0, (uint8_t) 0);

        DirStructType *cpm_dir;
//for (all i extent in block 0)
        for(int i=0; i<Extent_NO;i++ )
        {
                cpm_dir= malloc(sizeof(DirStructType));
                cpm_dir=mkDirStruct(i, block0);

                //if (dir is used)
                if(cpm_dir->status!=0xe5)
                {
                        for(int j=0;j<FILE_BLOCK_SIZE;j++)
                        {
                                // set freeList[i] == false for the used block
                                if(cpm_dir->blocks[j] != EMPTY_BLOCK)
                                {
                                        freeList[(int) cpm_dir->blocks[j]] = fal
                                }


                        }

                }
        }

}
```

3.4 printFreeList

This is a debugging function, which prints out the contents of the free list in 16 rows of 16, with each row prefixed by the 2-digit hex address of the first block in that row. Denote a used block with a *, a free block with a.

```
void printFreeList()
{
        printf("FREE_BLOCK_LIST: (* means in-use)\n");
        for(int i=0;i<FILE_BLOCK_SIZE;i++)
        {
                printf("%2x: ", i*FILE_BLOCK_SIZE);
```

```
                for(int offset=0; offset<FILE_BLOCK_SIZE;offset++)
                {
                        // print used block with *
                        if(!freeList[i*FILE_BLOCK_SIZE+offset]) printf("* ");

                        //print free block with .
                        else printf(". ");
                }
                printf("\n");
        }
}
```

3.5 cpmDir

This function prints the file directory to stdout. Each filename should be printed on its own line, with the file size, in base 10, following the name and extension, with one space between the extension and the size. If a file does not have an extension it is acceptable to print the dot anyway, e.g. "myfile. 234" would indicate a file whose name was myfile, with no extension and a size of 234 bytes. This function returns no error codes, since it should never fail unless something is seriously wrong with the disk

```
void cpmDir()
{
        uint8_t block0[BLOCK_SIZE];

        //read block0 from disk to in-memory
        blockRead(block0, (uint8_t) 0);

        printf("DIRECTORY LISTING\n");

        DirStructType *cpm_dir;

        //for all the extent referred by index in cpm_block0
        for(int i=0; i<Extent_NO;i++ )
        {
                cpm_dir= malloc(sizeof(DirStructType));
                cpm_dir=mkDirStruct(i, block0);
                int filesize = 0;
                //if cpm_dir->status is used
                if(cpm_dir->status!=0xe5)
                {
                        //count fully used file blocks
                        int NB=0;
                        for(int offset=0;offset<FILE_BLOCK_SIZE;offset++)
```

```
                        {
                                if(cpm_dir->blocks[offset] != EMPTY_BLOCK)
                                {
                                        NB++;
                                }
                        }
                //compute partially used file block size
                        int partial_flieblock = ((int) cpm_dir->RC)*128 + (int)c
                        //compute file length
                        filesize = (NB-1)*BLOCK_SIZE + partial_flieblock;


                //print file name and length from cpm
                        printf("%s.%s %d\n", cpm_dir->name,cpm_dir->extension,fi

                }
        }
}
```

3.6 checkLegalName

It is an internal function, returns true for legal name (8.3 format), false for illegal (name or extension too long, name blank, or illegal characters in name or extension)

```
bool checkLegalName(char *name)
{
        int i=0;
        for(;i<8 && name[i]!='.' && name[i]!='\0';i++)
        {
                //name can only have letter and digit
                if(name[i]<'0' || (name[i]>'9' && name[i] <'A') || (name[i] > 'Z
                {

                        return false;
                }
        }


        //check for too long file name
        if(i==8 && name[i]!='\0' && name[i]!='.')
        {
                return false;
        }
```

```
        //check for legal extension name
        else if(name[i]=='.')
        {
                i++;
                int j=0;

                for(;j<3 && name[i]!='\0';j++)
                {
                    //extension can only have letter and digit
                        if(name[i]<'0' || (name[i]>'9' && name[i]<'A') || (name
                        {
                                        return false;
                        }

                        i++;

                }
                //check for too long extension name
                if(name[i]!='\0' && j==3)
                {
                        return false;
                }
        }

        return true;
}
```

3.7 findExtentWithName

This is an internal function, which returns -1 for illegal name or name not found;
otherwise returns extent nunber 0-31

```
int findExtentWithName(char *name, uint8_t *block0)
{
        //split up the name into file_name, ext_name
        char file_name[9];
        char ext_name[4];

        //check legal file name, no blanks/punctuation/control chars
        if(!checkLegalName(name))
        {
                return -1;
        }

        int i=0;
        for(;i<8;i++)
```

```
        {
                file_name[i]=name[i];
                if(name[i]=='\0' || name[i]=='.') break;
        }

        //pad '\0' to file name
        file_name[i]='\0';

        //check legal ext_name
        if(name[i]=='.')
        {
                //ahead pointer to ext char
                i++;

                int extCount=0;
                for(;extCount<3;extCount++)
                {
                        ext_name[extCount]=name[i];
                        i++;
                }
                ext_name[extCount]='\0';
        }

        //for all dir entries in block0

        for(int j=0;j<Extent_NO;j++)
        {
                //obtain dir j from block0
                DirStructType *cpm_dir;
                cpm_dir=mkDirStruct(j, block0);

        //if dir->name==file_name
                if(!strcmp(cpm_dir->name,file_name))
                {
                    //if dir->status==valid
                        if(cpm_dir->status==0xe5) return -1;

                        //only return the index of the extent
                        return j;
                }

        }

        return -1;;
}
```

3.8 cpmDelete

The function deletes the file named name, and frees its disk blocks in the free list. This fuction returns -1 if file not found or illegal file name. Otherwise, returns 1 if file found and deleted successfully.

```
// return -1  if can't be deleted and 1 if delelted successfully
int  cpmDelete(char * name)
{
        uint8_t block0[BLOCK_SIZE];

        //load block0 from disk to in-memory
        blockRead(block0, (uint8_t) 0);
        int i;
        i=findExtentWithName(name,block0);
        //if file not found or illegal file name
        if(i<0)
        {
                //returns -1
                return i;
        }
        // else if file found
        else
        {

                DirStructType *cpm_dir;
                cpm_dir=mkDirStruct(i, block0);

                //mark 16 file blocks free in free list
                for(int j=0;j<FILE_BLOCK_SIZE;j++)
                {
                        // set freeList[i] == true for the used block
                        if(cpm_dir->blocks[j] != EMPTY_BLOCK)
                        {
                                freeList[(int) cpm_dir->blocks[j]] = true;
                        }

                        //set all file blocks as empty block so that further can
                        cpm_dir->blocks[j] = EMPTY_BLOCK;

                }

                //set status as unused
                block0[i*EXTENT_SIZE] = 0xe5;

                //write modified block0 to disk
                blockWrite(block0,(uint8_t) 0);
```

```
                //if extent deleted successfully return 1
                return 1;
        }
}
```

3.9 cpmRename

This function reads directory block, modifies the extent for file named oldName with newName, and write to the disk. This fuction returns 0 if the extent can be modified successfully. Otherwise, returns -1 if extent can't be modified for illegal name or requested extent doesn't exist.

```
// modify the extent for file named oldName with newName, and write to the disk
//returns -1 if can't be modified or 0 if modified successfully
int cpmRename(char *oldName, char * newName)
{
//if newName is illegal return -1
        if(!checkLegalName(newName)) return -1;

        uint8_t *block0=malloc(BLOCK_SIZE);

//load block0 from disk to in-memory
        blockRead(block0, (uint8_t) 0);

        int j = findExtentWithName(oldName,block0);

//if extent with oldName doesn't exist return -1
        if(j<0) return -1;

//else if extent with oldName found rename it
        else
        {

                //write extent with oldName from in-memory Block0 to DirStruct
                DirStructType *cpm_dir;
                cpm_dir=mkDirStruct(j, block0);


    //modify file_name and extension of the extent of block0 according to newNam

    //Step1: separate fileName and extension of newName
                char file_name[9];
                char ext_name[4];
                int i=0;
                for(;i<8;i++)
```

```
        {
                file_name[i]=newName[i];
                if(newName[i]=='\0' || newName[i]=='.') break;
        }

        //pad '\0' to file name
        file_name[i]='\0';

        if(newName[i]=='.')
        {
                //ahead pointer to ext char
                i++;

                int extCount=0;
                for(;extCount<3;extCount++)
                {
                        ext_name[extCount]=newName[i];
                        i++;
                }
                ext_name[extCount]='\0';
        }


//Step2: modify oldName to newName in DirStruct
        int o=0;
        while(file_name[o]!='\0' && file_name[o] != '.')
        {
                cpm_dir->name[o]=file_name[o];
                o++;
        }

        //if name length<8 pad with ' '
        if(o<8)
        {
                while(o<8)
                {
                        cpm_dir->name[o]=' ';
                        o++;
                }
        }

        int c=0;
        while(ext_name[c]!='\0')
        {
                cpm_dir->extension[c]=ext_name[c];
                c++;
```

```
                }

                //if extension length<3 pad with ' '
                if(c<3)
                {
                        while(c<3)
                        {
                                cpm_dir->extension[c]=' ';
                                c++;
                        }
                }

    //Step3: write DirStruct to in-memory block0
        writeDirStruct(cpm_dir, j, block0);
        //Step4: write modified in-memory block0 to disk
                blockWrite(block0,0);

                return 0;
        }

}
```

4 Lesson Learned

1. To design a simple file system
2. To explain the function of file systems
3. To learn and implement directory structures
4. To implement block allocation and free-block algorithms
5. To debug C program in Linux
6. To strengthen debugging skills
7. To improve software development skills
8. To enhance operating systems research skills
9. In printFreeList(), it did't print * for the first block of block0. Then I realized that I just consider blocks of used Extents to make the freeList false. But block 0 is not used by users. So, I explicictly had to mark it as used.