# CPS801 – Operating System

## Project

**Warren Pinto: 500396119**

**April 10, 2014**

**CPS801 Project**

**2A)**

Some assumptions were made in the file system was that the initial HDD space was empty, and then populated with the initial files. For any additional write statements, the file will be written to the HDD only if there is enough space (400 blocks). If the space is full and a file is requested to be written, then it must wait for another file to be deleted to free up space. File deletion deletes all the files associated with that file number.

Another write assumption is that if a file does not exist, a new file is written at the end of the data structure.

**B)**

The data-structure that was used was a linked list. To populate the linked list initially, the file information has to be read from the fileList.txt file (made from part1.c). Each file is placed in a node of the linked list, along with its information such as file size, number of blocks and block number.

For a write request: if there is enough space in the node (40blocks for each file), then block size will be added. If there is insufficient space in the node, a new node is created just next to the previous node. This implementation makes it easier and faster to locate/find the nodes of the same file number that have been spread out in the data-structure.

To implement a directory system, the file nodes could have information stating which directory the file belongs to. A separate data-structure could be made that connects different directories together such as a linked-list. Another implementation of the directory system could be i-nodes.

**Part1.c**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

void initFiles();

void listFiles();

void createFile();

void readStruct();

struct file{

int fileNum;

int blockSize;

int size;

struct file \*next;

};

struct file \*fileTableHead = (struct file\*)NULL;

struct file \*fileTableCurrent = (struct file \*)NULL;

struct file \*fileTable\_ptr = (struct file \*)NULL;

int main(int argc, char \*argv[]){

if(argc > 1){

if(0 == strcmp(argv[1], "-r"))

listFiles();

else if(0 == strcmp(argv[1], "-c"))

createFile();

else if(0 == strcmp(argv[1], "-init"))

initFiles();

else if(0 == strcmp(argv[1], "-d"))

deleteFile(argv[2]);

else if(0 == strcmp(argv[1], "-rn"))

renameFile(argv[2], argv[3]);

else if(0 == strcmp(argv[1], "-rs"))

readStruct();

}

else{

printf("Not enough Arguments.\n");

}

exit(0);

}

void initFiles(){

srand(time(NULL));

/\*generate 10 files with random size. 1 block = 512 bytes. 1 file 1-40 blocks\*/

FILE \*file;

file = fopen("filelist.txt", "w+");

int i = 0;

/\*Create new file table head\*/

fileTable\_ptr = (struct file\*)malloc(sizeof(struct file));

fileTable\_ptr->fileNum=0;

fileTable\_ptr->blockSize=0;

fileTable\_ptr->size=0;

fileTableHead = fileTable\_ptr;

fileTableCurrent = fileTableHead;

for(i = 0; i < 10; i++){

int size = rand() % 40 + 1; /\*Generate size file blocks 1-40\*/

int bytes = size\*512; /\*calculate bytes per block\*/

char buffer[20];

char buff2[10];

strcpy(buffer, "file\_");

sprintf(buff2, "%d", i);

strcpy(buffer, buff2);

/\*make new file node\*/

struct file \*newFile\_ptr = (struct file\*)malloc(sizeof(struct file));

newFile\_ptr->fileNum=i;

newFile\_ptr->blockSize=size;

newFile\_ptr->size=bytes;

fileTableCurrent->next = newFile\_ptr;

fileTableCurrent = fileTableCurrent->next;

fprintf(file,"%d %d %d %s\n", i, size, bytes, buffer);

}

fclose(file);

fileTableHead = fileTableHead->next;

fileTableCurrent = fileTableHead;

while(fileTableCurrent->next != NULL){

printf("NODE: %d", fileTableCurrent->fileNum);

fileTableCurrent = fileTableCurrent->next;

}

}

void listFiles(){

FILE \*file;

file = fopen("filelist.txt", "r");

int \*f\_num, \*f\_block, \*f\_size;

char \*f\_name;

while(fscanf(file, "%d %d %d %s", &f\_num, &f\_block, &f\_size, &f\_name) != EOF){

printf("file: %d\n", f\_num);

}

fclose(file);

}

void createFile(){

srand(time(NULL));

FILE \*file;

file = fopen("filelist.txt", "r");

int \*file\_num=0;

int \*file\_block, \*file\_size;

char \*file\_name;

int i=0;

while(fscanf(file, "%d %d %d %s", &file\_num, &file\_block, &file\_size, &file\_name) != EOF){i++;}

fclose(file);

file = fopen("filelist.txt", "a+");

int size = rand() % 40 + 1; /\*Generate size file blocks 1-40\*/

int bytes = size\*512; /\*calculate bytes per block\*/

char buffer[20];

char buff2[10];

strcpy(buffer, "file\_");

sprintf(buff2, "%d", i);

strcpy(buffer, buff2);

fprintf(file,"%d %d %d %s\n", i, size, bytes,buffer);

fclose(file);

}

void renameFile(char \*file\_name, char \*replace){

FILE \*file;

file = fopen("filelist.txt", "r");

int \*file\_num=0;

int \*file\_block, \*file\_size;

int i=0;

char \*f\_name;

while(fscanf(file, "%d %d %d %s", &file\_num, &file\_block, &file\_size, &f\_name) != EOF){

i++;

if(0==strcmp(&f\_name, file\_name))

break;

}

fclose(file);

FILE \*ff,\*ff2;

ff = fopen("filelist.txt", "r");

ff2 = fopen("filelist2.txt", "w");

while(fscanf(ff, "%d %d %d %s", &file\_num, &file\_block, &file\_size, &f\_name) != EOF){

if(file\_num == (i-1))

fprintf(ff2, "%d %d %d %s\n", file\_num, file\_block, file\_size, replace);

else

fprintf(ff2, "%d %d %d %s\n", file\_num, file\_block, file\_size, &f\_name);

}

fclose(ff);

fclose(ff2);

system("mv filelist2.txt filelist.txt");

}

void deleteFile(char \*file\_name){

FILE \*file;

file = fopen("filelist.txt", "r");

int \*file\_num=0;

int \*file\_block, \*file\_size;

int i=0;

char \*f\_name;

while(fscanf(file, "%d %d %d %s", &file\_num, &file\_block, &file\_size, &f\_name) != EOF){

i++;

if(0==strcmp(&f\_name, file\_name))

break;

}

fclose(file);

FILE \*ff,\*ff2;

ff = fopen("filelist.txt", "r");

ff2 = fopen("filelist2.txt", "w");

while(fscanf(ff, "%d %d %d %s", &file\_num, &file\_block, &file\_size, &f\_name) != EOF){

if(file\_num == (i-1)){

}

else

fprintf(ff2, "%d %d %d %s\n", file\_num, file\_block, file\_size, &f\_name);

}

fclose(ff);

fclose(ff2);

system("mv filelist2.txt filelist.txt");

}

void readStruct(){

/\*reset current pointer to head of table\*/

fileTableCurrent = fileTableHead;

printf("PASS1: %d\n", fileTableCurrent->fileNum);

while(fileTableCurrent != NULL){

printf("PASS2\n");

printf("STRUCT FILE: %d", fileTableCurrent->fileNum);

printf("pass3\n");

fileTableCurrent = fileTableCurrent->next;

printf("pass4\n");

}

}

**Part2.c**

#define \_REENTRANT

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/time.h>

#include <unistd.h>

#include <time.h>

#include "semaphore.c"

#define MAX 10

#define MAX\_READS 2

void readT();

void printT();

void readFileList();

void \* threadRequest();

void performRequests();

void readF(int fn);

void writeF(int fNum, int newBlockSize);

void deleteF(int fNum);

typedef int semaphore;

typedef int rwd\_sempahore;

int sem1;

int reads;

int sem2;

int space;

struct file {

int fileNum;

int blockSize;

int size;

int blockNum;

struct file \*next;

};

struct file \*fileHead = NULL;

struct file \*fileCurrent = NULL;

void main(int argc, char \*argv[]){

int \*option;

readT();

while(option != -1){

printf("[1] Perform Requests\n");

printf("[2] Print File List\n");

printf("[-1] EXIT\n");

scanf("%d", &option);

if(option == 1)

performRequests();

else if(option == 2)

printT();

else if(option == -1)

printf("Program exiting...\n");

else

printf("Invalid input.");

}

}

/\*File pre-loading phase. Pre-load all existing files from part1.c.\*/

void readT(){

FILE \*f;

f = fopen("filelist.txt", "r");

int \*fnum, \*fbsize, \*fsize;

char \*f\_name;

struct file \*headFile = (struct file \*)malloc(sizeof(struct file));

headFile->fileNum = 0;

headFile->blockSize = 0;

headFile->size = 0;

headFile->blockNum = 0;

headFile->next = NULL;

fileHead = headFile;

fileCurrent = fileHead;

int bNum = 0;

while(fscanf(f, "%d %d %d %s", &fnum, &fbsize, &fsize, &f\_name) != EOF){

bNum = fileCurrent->blockNum + fileCurrent->blockSize;

struct file \*newFile = (struct file \*)malloc(sizeof(struct file));

newFile->fileNum = fnum;

newFile->blockSize = fbsize;

newFile->size = fsize;

newFile->blockNum = bNum;

space+=(newFile->blockSize);

printf("HDD Space: %d\n", space);

fileCurrent->next = newFile;

fileCurrent = fileCurrent->next;

}

}

/\*Prints the files in the system.\*/

void printT(){

fileCurrent = fileHead;

space = 0;

while(fileCurrent->next != NULL){

printf("FILE: %d [%d block #] [%d blocks] [%d size]\n", fileCurrent->fileNum, fileCurrent->blockNum, fileCurrent->blockSize, fileCurrent->size);

space+=(fileCurrent->blockSize);

fileCurrent = fileCurrent->next;

}

printf("HDD Space: %d\n", space);

}

void readFileList(){

fileCurrent = fileHead;

while(fileCurrent != NULL){

printf("%d", fileCurrent->fileNum);

fileCurrent = fileCurrent->next;

}

}

/\*Create threads that will perform operations.\*/

void performRequests(){

pthread\_t thread\_id[MAX];

int status, \*p\_status = &status;

srand(time(NULL));

int i;

int argc = 1;

sem1 = createSem();

for(i = 0; i < MAX; i++){

if(pthread\_create(&thread\_id[i], NULL, threadRequest, NULL) > 0)

printf("THREAD FAILED TO CREATE\n");

}

for(i = 0; i < MAX; i++){

if(pthread\_join(thread\_id[i], (void \*\*) p\_status) > 0)

printf("THREAD FAILED TO CREATE\n");

}

printf("Thread returned %d returns %d\n", thread\_id[i], status);

}

/\*Method that will handle all thread related actions such as

file read, write, delete. Actions are controlled by semaphores.

Only one theread may perform an operation.\*/

void \* threadRequest(){

int randFile = rand() % 8 + 1;

int randReq = rand() % 3 + 1;

char op;

down(sem1);

//enter critical region

FILE \*f;

f = fopen("fileoperations.txt", "a+");

if(randReq == 1){

op = 'R';

//run read thread.

readF(randFile);

}

else if(randReq == 2){

op = 'W';

int randNewBlocks = rand() % 40 + 1;

int randNewBytes = randNewBlocks \* 512;

printf("\*\*WRITING TO FILE %d [+%d blocks]\n\*\*", randFile, randNewBlocks);

writeF(randFile, randNewBytes);

}

else{

op = 'D';

printf("DELETE FILE %d\n", randFile);

deleteF(randFile);

}

fprintf(f, "%d %c\n", randFile, op);

fclose(f);

//exit critical region

up(sem1);

return (void \*) NULL;

}

/\*Handles read statements\*/

void readF(int fn){

FILE \*f;

f = fopen("fileLog.txt", "a+");

fileCurrent = fileHead;

int fileReadStatus = 0;

/\*Read file system linked list.\*/

while(fileCurrent->next != NULL){

if(fn == fileCurrent->fileNum){

fileReadStatus = 1;

fprintf(f, "%d\t R\t [%d blocks] [%d size] Success\n", fileCurrent->fileNum, fileCurrent->blockSize, fileCurrent->size);

break;

}

fileCurrent = fileCurrent->next;

}

/\*If file does not exist, output failed log to log file.\*/

if(fileReadStatus == 0)

fprintf(f, "%d\t R\t [%d blocks] [%d size] FAILED - file not found\n", fileCurrent->fileNum, fileCurrent->blockSize, fileCurrent->size);

fclose(f);

}

/\*Handles write statements\*/

void writeF(int fNum, int newFbytes){

int newFblocks = newFbytes / 512;

int bNum = 0;

fileCurrent = fileHead;

int fileExists = 0;

int bn = 0;

FILE \*f;

f = fopen("fileLog.txt", "a+");

while(fileCurrent->next != NULL){

space+=(fileCurrent->blockSize);

/\*check whether the HDD is fully occupied. If it is, output

to log file and stop file write.\*/

if(space >= 400){

fprintf(f, "%d\t W\t [%d blocks] [%d size] FAILED - HDD FULL.\n", fNum, newFblocks, newFbytes);

break;

}

if(fNum == fileCurrent->fileNum){

fileExists = 1;

int currBlock = fileCurrent->blockSize + newFblocks;

/\*test whether the current file has all its blocks

occupied when adding new blocks. If it exceeds 40,

add new block to the end of file.

Add pointer of current block to the end block.

\*/

int blockRemaining = 40 - fileCurrent->blockSize;

if((fileCurrent->blockSize + newFblocks) > 40){

//create new block at end of stucture if not enough space.

struct file \*newFile = (struct file \*)malloc(sizeof(struct file));

newFile->fileNum = fNum;

newFile->blockSize =newFblocks;

newFile->size = newFblocks\*512;

newFile->blockNum = space;

fprintf(f, "%d\t W\t [%d blocks] [%d size] Success - New node created.\n", fileCurrent->fileNum, fileCurrent->blockSize, fileCurrent->size);

struct file \*temp = fileCurrent->next;

fileCurrent->next = newFile;

fileCurrent = fileCurrent->next;

fileCurrent->next = temp;

break;

}

else if (blockRemaining > 0 && newFblocks <= blockRemaining){

//add new block at current node if enough space.

printf("adding new blocks...%d\n", newFblocks);

fileCurrent->blockSize = currBlock;

fileCurrent->size = currBlock\*512;

fileCurrent->blockNum = space;

fprintf(f, "%d\t W\t [%d blocks] [%d size] Success\n", fileCurrent->fileNum, fileCurrent->blockSize, fileCurrent->size);

}

}

bn = fileCurrent->blockNum + fileCurrent->blockSize;

fileCurrent = fileCurrent->next;

}

/\*check whether a file actually exists in the system.

If it doesnt, create a new file IF space is available.\*/

if(fileExists == 0){

struct file \*newFile = (struct file \*)malloc(sizeof(struct file));

newFile->fileNum = fNum;

newFile->blockSize =newFblocks;

newFile->size = newFblocks\*512;

newFile->blockNum = space;

fileCurrent->next = newFile;

fprintf(f, "%d\t W\t [%d blocks] [%d size] FAILED - creating new file...\n", fileCurrent->next->fileNum, fileCurrent->next->blockSize, fileCurrent->next->size);

fprintf(f, "%d\t W\t [%d blocks] [%d size] Success - created new file.\n", fileCurrent->next->fileNum, fileCurrent->next->blockSize, fileCurrent->next->size);

}

fclose(f);

}

/\*delete a file completely.

Basic linked list deletion.\*/

void deleteF(int fNum){

FILE \*f;

f = fopen("fileLog.txt", "a+");

int fileDeleteStatus = 0;

fileCurrent = fileHead;

while(fileCurrent->next != NULL){

if(fileCurrent->next->next != NULL){

if(fileCurrent->next->fileNum == fNum){

fileDeleteStatus = 1;

fprintf(f, "%d\t D\t [%d blocks] [%d size] Success\n", fileCurrent->next->fileNum, fileCurrent->next->blockSize, fileCurrent->next->size);

fileCurrent->next = fileCurrent->next->next;

}

}

fileCurrent = fileCurrent->next;

}

/\*check whether a file actually exists in the system.

If it doesnt, output to log (no file found).\*/

if(fileDeleteStatus == 0)

fprintf(f, "%d\t D\t [0] [0] FAILED - No such file...\n", fNum);

fclose(f);

}