ECE 421 Assignment 2 – Progress Check

Brandon Rubio 2/18/2023

File “test.b” in /home/students/bur2/OS/Assignments/test

Overview:

* Current program is a modification of the completed single block file system done in class
* Modified format(), mount(), and dismount() functions to accommodate current design plan
* Tasks that still need to be completed:
  + Adding file to directory
    - I think It needs a way to search for the path of the current working directory and parent directory
    - Need to modify lookup() function I believe
    - This also needs to incorporate finding and allocating free memory blocks from disc
  + Removing file from directory
  + The iosb and the higher level functions that come with it

I will attach my assignment 2 current design plan and all the information I have about creating a file system. There are still many sections I need to flesh out when it comes to the iosb but my primary focus right now is the core of the file system. In particular, could you look over my design for the header block specifications of a directory and a file? I’m not entirely sure on how to implement the storage of directory entries.

**Assignment 2 Information and Design:**

Notes: Class 6: intro to indexed file systems (explains level 1 as well)

Class 4: intro to iosb

Class 9: expands on iosb

Class 10: framework for file system

The system is set up by default to have one drive with 6000 blocks

a block is 512 bytes

a directory is just a special kind of file, a block can be dedicated as a directory (such as rootdir)

better system than the single block files:

globals:

superblock = vec(128);

rootdir = vec(128);

stackView = vec(256);

mounting:

copy superblock, rootdir, and stackView into memory

- top can range from 0 to discSize - freeListTotBlocks + 2

- therefore:

- block number = top / 128 + 1 ( + 1 to exclude superblock, which is at block 0)

- top position in that block = top rem 128

Formatting the disk:

- superblock (block 0)

- contains the position (block) of the rootdir

- top of stack position (relative to end of free list, set to 0)

- end of stack position (discSize - 1 - freeListTotBlocks + 2)

- should have a free list, shows which blocks are free to use

- 128 numbers can be stored in a single block

- (disc size + 127) / 128 = number of blocks needed for free list

- adds 127 so that integer division provides the correct number of blocks

- without it it would be roof(disc size/128)

- to be a little more efficient with storage we could say that there must always

be 2 blocks dedicated for rootdir and superblock

Therefore the new equation could be (disc size - 2 + 127) / 128

That way there wont be an entire block's worth of space added to the free list

total blocks when the result of (disc size - 2 + 127) mod 128 is 1 or 2

- set the rootdir right after the free list

- store every free block number in the free list

- therefore blocks 0, 1 through freeListTotBlocks, and freeListTotBlocks + 1 (rootdir) will be taken initially

- blocks from freeListTotBlocks + 2 to discSize - 1 will be free

- the idea behind this is to use the free list as a stack

- pop from the top of stack if you want to use that block

- add back to top of stack if you want to free that block

- finally zero fill the contents of rootdir

enter: (creating an entry to a directory AKA creating a new file)

- look through free list and find first free entry

- if no free entry return an error

- otherwise update that entry to be taken

- move top to top + 1

- set limit so that top does not move past the end

note: when freeing an entry

- move top to top - 1 and overide the value at top - 1 with the entry you are freeing

- works since the file header and directories will keep track of what blocks hold the

file's data

- set limit so that i does not move past the beginning

header block design:

-represents a whole file

-112 bytes (arbitrary) to store metadata

-leaves 400 bytes to store block numbers (100 of them) (for files)

for the metadata:

- type (1 byte): file or directory (special kind of file)

- how a pointer in the data section is interpreted is determined by the

header block type

- name of file/directory (8 bytes)

- file/directory creation time (8 bytes)

- last modified time (8 bytes)

- last accessed time (8 bytes)

- protections (currently dont know how to implement)

- level (for multi-level file indeces, not neccessary to implement at this time)

for actual data:

- total space available (for now): 512 -33 = 479 bytes

- each pointer is one word (4 bytes) (can represent blocks 0 to 2^32 - 1)

- total pointers that can be created: 479 / 4 = 119 with 3 bytes left over

file data:

- pointers to blocks (NOT header blocks) that contain file data

- pointers to pointer blocks can be added later (mulit level file indeces)

directory data:

Design 1:

- only store pointers to header blocks

Advantages:

- allows for more files to be stored per directory

- results in some extra storage space

- directories and files would have the same metadata

- make things less complex

Disadvantages:

- Lets say a user would like to compare the names of the files in the directory.

The system would have to search through every pointer in the directory and

load each header block that it points to in order to read that entry's data.

- I believe this would slow down the file system considerably

- why would it slow it down?

- isn't it reading/writing from disc managed by devctl

- how long does devctl take?

Alternatively:

Design 2:

- store pointers to header blocks

- store metadata of files as well

- would need 32 bytes per file entry to store

- 479 / 32 = only 14 entries per directory

iosb design:

basic functions necessary:

open

-tape for reading

-tape for writing

-disc for reading

-disc for writing

-tty for reading

-readch

-tty for writing

-writech

close

-every type form open

-like close(f)

iosb is a vector

-stores functions:

-readch

-writech

-close(f)

can thnk of it like a hadrware instruction

where the iosb contains info in a similar format to

opcode, register, modes, etc.

iosb needs to be extended to include:

-when dealing with tapes and buffers, need to create

a new buffer for each new file open

-also when you open a file, allocate a block of memory to

store the header block

-pos inside of buffer (currently viewing this pos)

-size of buffer (how much is currently stored in said buffer)

-pos and size will help fill the buffer with the blocks of data

you want to read, can be used for tapes or discs

-keep in mind you have to create pos and size for every buffer

-length left

- this should be obtained from the header block of the file

- shows how much data is left from in the file

- once it is less than 512 bytes set the size of the buffer to

only accept what is left

**Program as found in file “test.b”:**

import "io"

manifest {

sbPos = 0 //super block is located in block zero

}

//these vectors are copied from the mounted disk

let superblock = vec(128);

let rootdir = vec(128);

let stackView = vec(256);

//max refers to maximum index to iterate to

let strncpy(dest, src, max) be{

let i = 0, ended = false;

while i <= max do

{ test ended then

byte i of dest := 0

else

{ let c = byte i of src;

byte i of dest := c;

if c = 0 then

ended := true }

i +:= 1 }

}

let strncmp(a, b, max) be {

for i = 0 to max do

{ if byte i of a <> byte i of b then

resultis false;

if byte i of a = 0 then

resultis true }

resultis true

}

// checks if two strings are equal

let equals(s, t) be {

let i = 0;

while byte i of s = byte i of t do

{ if byte i of s = 0 then

resultis true;

i +:= 1 }

resultis false

}

//s will be interpreted as a string

let outsn(s, max) be { //prints a slected amount of characters from a string

for i = 0 to max do

test byte i of s = 0 then

return

else

outch(byte i of s)

}

let format() be { //clears and formats the entire disc

let buffer = vec(128), r; //512 byte buffer, r for result of devctl functions

let freeListSize, discSize; // in blocks

let rootdirPos;

// used to initialize superblock

let top, end;

// used for creating the free list

let pos = 0, storageBlock = 1;

discSize := devctl(DC\_DISC\_CHECK, 1);

if discSize <= 0 then{

out("format error, disc unavailable\n");

return;

}

out("format the disc\n");

// clear the disc

for i = 0 to 127 do { // set buffer to 0

buffer ! i := 0;

}

r := devctl(DC\_DISC\_WRITE, 1, 0, discSize, buffer);

if r < 0 then {

out("format error %d, unable to clear disc\n", r);

return;

}

freeListSize := (discSize + 127) / 128;

rootdirPos := freeListSize + 1;

top := 0;

end := discSize - 1 - freeListSize + 2;

// initialize superblock

buffer ! 0 := rootdirPos;

buffer ! 1 := top;

buffer ! 2 := end;

r := devctl(DC\_DISC\_WRITE, 1, sbPos, 1, buffer);

if r < 0 then {

out("format error %d, unable to initialize superblock\n", r);

return;

}

//create free list

for blockNumToStore = freeListSize + 2 to discSize - 1 do {

buffer ! pos := blockNumToStore;

pos +:= 1;

test pos = 127 then {

if storageBlock > freeListSize then {

out("format error, free list storage overflow\n");

return;

}

devctl(DC\_DISC\_WRITE, 1, storageBlock, 1, buffer);

pos := 0;

storageBlock +:= 1;

}

else { // if there is not enough to fill the buffer write the rest in the given block

devctl(DC\_DISC\_WRITE, 1, storageBlock, 1, buffer);

}

}

}

// mount copies the super block, root directory, and stack view from disc to their corresponding vectors

let mount() be {

let r; //result of devctl function

let freeListSize, discSize;

let top, blockNumber;

out("mount the disk\n");

//load superblock

r := devctl(DC\_DISC\_READ, 1, sbPos, 1, superblock);

if r < 0 then{

out("mount error %d, unable to load superblock\n", r);

return;

}

//load rootdir

r := devctl(DC\_DISC\_READ, 1, superblock ! 0, 1, rootdir);

if r < 0 then{

out("mount error %d, unable to load rootdir\n", r);

return;

}

//load stackView

discSize := devctl(DC\_DISC\_CHECK, 1);

if discSize <= 0 then {

out("mount error, disc unavailable\n");

return;

}

freeListSize := (discSize + 127) / 128;

top := superblock ! 1;

blockNumber := top / 128 + 1;

test freeListSize < 2 then { // assuming that freeListSize block size is at least 1

r := devctl(DC\_DISC\_READ, 1, blockNumber, 1, stackView);

if r < 0 then {

out("mount error %d, unable to read one block into stackView\n", r);

return;

}

}

else { // if at least two blocks are dedicated to the free list

r := devctl(DC\_DISC\_READ, 1, blockNumber, 1, stackView);

if r < 0 then {

out("mount error %d, unable to read frist (possibly second) block into stackView\n", r);

return;

}

test blockNumber + 1 <= freeListSize then { //if a block in at the front or middle is read

r := devctl(DC\_DISC\_READ, 1, blockNumber + 1, 1, stackView ! 128);

if r < 0 then {

out("mount error %d, unable to read second block into stackView\n", r);

return;

}

}

else { // if a block at the end is read

r := devctl(DC\_DISC\_READ, 1, blockNumber - 1, 1, stackView ! 128);

if r < 0 then {

out("mount error %d, unable to read (technically) the first block into stackView\n", r);

return;

}

}

}

}

let dismount() be {

let r; // result of devctl

let freeListSize, discSize; // in blocks

let blockNumber, top, topPos;

let temp = vec(128);

out("dismount the disc\n");

//write back to super block

r := devctl(DC\_DISC\_WRITE, 1, sbPos, 1, superblock);

if r < 0 then {

out("dismount error %d, unable to write to super block\n", r);

}

//write back to root directory

r := devctl(DC\_DISC\_WRITE, 1, superblock ! 0, 1, rootdir);

if r < 0 then {

out("dismount error %d, unable to write to root directory\n", r);

}

//write back stack view

discSize := devctl(DC\_DISC\_CHECK, 1);

if discSize <= 0 then {

out("dismount error, disc unavailable\n");

return;

}

freeListSize := (discSize + 127) / 128;

top := superblock ! 1;

blockNumber := top / 128 + 1;

topPos := top rem 128;

r := devctl(DC\_DISC\_READ, 1, blockNumber, 1, temp);

if r < 0 then {

out("dismount error %d, unable to read to temp buffer\n", r);

}

for i = 0 to 255 do {

if stackView ! i = temp ! topPos then {

test i <= 127 then {

r := devctl(DC\_DISC\_WRITE, 1, blockNumber, 2, stackView);

if r < 0 then {

out("dismount error %d, unable to write back stackView\n", r);

return;

}

return;

}

else {

r := devctl(DC\_DISC\_WRITE, 1, blockNumber - 1, 2, stackView);

if r < 0 then {

out("dismount error %d, unable to write back stackView (b-1)\n", r);

return;

}

return;

}

}

}

out("dismount error, unable to find top in stackView\n");

return;

}

let enter(name, blocknumber) be { // add an entry to a given directory

out("add \"%s\" -> %d to the directory\n", name, blocknumber);

for ptr = 0 to 127 by 4 do

if rootdir ! ptr = 0 then

{ strncpy(rootdir + ptr, name, 7);

rootdir ! (ptr + 2) := seconds();

rootdir ! (ptr + 3) := blocknumber;

resultis ptr }

resultis -1

}

let lookup(name) be // searches directory, returns block number

{ out("lookup \"%s\" in the directory\n", name);

for ptr = 0 to 127 by 4 do

if rootdir ! ptr <> 0 then

if strncmp(name, rootdir + ptr, 7) then

resultis rootdir ! (ptr + 3);

resultis -1 }

let remove(name) be // remove a directory entry

{ out("remove \"%s\" from the directory\n", name);

for ptr = 0 to 127 by 4 do

if rootdir ! ptr <> 0 then

if strncmp(name, rootdir + ptr, 7) then

{ rootdir ! ptr := 0;

rootdir ! (ptr + 1) := 0;

rootdir ! (ptr + 2) := 0;

rootdir ! (ptr + 3) := 0;

resultis 1 }

resultis -1 }

let print.dir() be

{ let v = vec(7);

out("list the directory\n");

for ptr = 0 to 127 by 4 do

if rootdir ! ptr <> 0 then

{ outsn(rootdir + ptr, 7);

datetime(rootdir ! (ptr + 2), v);

out(", %04d-%02d-%02d %02d:%02d:%02d, block %d\n",

v ! 0, v ! 1, v ! 2, v ! 4, v ! 5, v ! 6, rootdir ! (ptr + 3)) } }

let update.stackView() be

{

}

let find.free.block() be

{ let b = superblock ! sb.firstfree;

out("find a free block\n");

superblock ! sb.firstfree +:= 1;

resultis b }

let releaseBlock(blocknumber) be

{

}

let write(name, contents) be // contents is everything all in one string

{ let bn = find.free.block(), r;

out("create file \"%s\" with given content\n", name);

enter(name, bn);

r := devctl(DC\_DISC\_WRITE, 1, bn, 1, contents);

if r < 0 then

out("write failed at write/1\n");

resultis r }

let read(name, buffer) be // read entire contents into buffer

{ let blocknumber = lookup(name), r;

out("read file \"%s\" into buffer\n", name);

if blocknumber < 0 then

{ out("read: lookup failed\n");

resultis -1; }

r := devctl(DC\_DISC\_READ, 1, blocknumber, 1, buffer);

if r < 0 then

out("read failed at read/1\n");

resultis r }

let read.line(string, size) be

{ let length = 0;

size -:= 1;

while true do

{ let c;

if length = size then

{ byte length of string := 0;

resultis length; }

c := inch();

if c = '\n' then

{ byte length of string := 0;

resultis length; }

byte length of string := c;

length +:= 1; } }

let read.lines(string, size) be

{ let length = 0, last.was.newline = false, maybe.ending = false;

size -:= 1;

while true do

{ let c;

if length = size then

{ byte length of string := 0;

resultis length; }

c := inch();

if maybe.ending /\ c = '\n' then

{ byte length - 1 of string := 0;

resultis length }

maybe.ending := last.was.newline /\ c = '\*';

last.was.newline := c = '\n';

byte length of string := c;

length +:= 1; } }

let start() be

{ let buffer = vec(129);

while true do

{ out("> ");

read.line(buffer, 512);

if byte 0 of buffer = 0 then

loop;

if buffer %equals "exit" then

finish;

test buffer %equals "format" then

format()

else test buffer %equals "mount" then

mount()

else test buffer %equals "dismount" then

dismount()

else test buffer %equals "findfree" then

out("found free block %d\n", find.free.block())

else test buffer %equals "enter" then

{ let n, r;

out("file name: ");

read.line(buffer, 512);

out("block number: ");

n := inno();

r := enter(buffer, n);

out("returned %d\n", r) }

else test buffer %equals "lookup" then

{ let n;

out("file name: ");

read.line(buffer, 512);

n := lookup(buffer);

out("block number = %d\n", n) }

else test buffer %equals "rm" then

{ let r;

out("file name: ");

read.line(buffer, 512);

r := remove(buffer);

out("returned %d\n", r) }

else test buffer %equals "ls" then

print.dir()

else test buffer %equals "write" then

{ let content = vec(129);

out("file name: ");

read.line(buffer, 512);

out("content (lone \* at end):\n");

read.lines(content, 512);

write(buffer, content) }

else test buffer %equals "read" then

{ let content = vec(129);

out("file name: ");

read.line(buffer, 512);

read(buffer, content);

out("content = \"%s\"\n", content) }

else test buffer %equals "help" then

out("exit format mount dismount findfree enter lookup rm ls write read help\n")

else

out("bad command \"%s\"\n", buffer) } }