Assignment 4 – Create Simple Yet Functional File System

CMPT 300 – Operating Systems – Instructor: Nate Payne

Please submit a zip folder with the following naming conventions to canvas:

LastName_FirstName_StudentNumber_Assig4

This file should include all code, and a text file called answers.txt that includes answers to all questions. There may be questions for both part 1 and part 2, so make sure that you address all questions.

Part 1 - I/O and Filesystems

Welcome to LMP1, the first long MP. LMP1 is the first stage of a project aimed at creating a simple yet functional networked filesystem. In this MP, you will learn about and use POSIX file system calls, while subsequent LMPs will introduce memory management, messaging, and networking functionality. If you implement all parts of this MP correctly, you will be able to reuse your code for future MPs.

This first LMP concentrates on the file I/O portion of the project. Specifically, you will implement a custom filesystem and test its performance using a filesystem benchmark. A benchmark is an application used to test the performance of some aspect of the system. We will be using Bonnie, a real filesystem benchmark, to test various performance aspects of the filesystem we implement.

LMP1 consists of four steps:

1. Read the code; run the Bonnie benchmark and the LMP1 test suite.

2. Implement Test Suite 1 functionality, encompassing basic file I/O operations.

3. Implement Test Suite 2-4 functionality (directory operations, file creation/deletion, and recursive checksumming).

4. Modify Bonnie to use your client-server file I/O methods.

Code structure

The code for this project is structured according to the client-server model. The client code (filesystem benchmark) will interact with the server (filesystem) only through interface functions defined in fileio.h:

```
int file_read(char *path, int offset, void *buffer, size_t bufbytes);
int file_info(char *path, void *buffer, size_t bufbytes);
int file_write(char *path, int offset, void *buffer, size_t bufbytes);
```

```
int file_create(char *path,char *pattern, int repeatcount);
int file_remove(char *path);

int dir_create(char *path);
int dir_list(char *path,void *buffer, size_t bufbytes);

int file_checksum(char *path);
int dir_checksum(char *path);
```

These functions represent a simple interface to our filesystem. In Steps 2 and 3 of this MP, you will write the code for functions implementing this interface, replacing the stub code in fileio.c. In Step 4, you will modify a Bonnie method to use this interface, rather than calling the normal POSIX I/O functions directly. The purpose of Step 4 is to help test our implementation.

Step 1: Understanding the code

1. Compile the project, execute Bonnie and the test framework.

Note: you may need to add execute permissions to the .sh files using the command "chmod +x *.sh".

Try the following:

```
make
./lmp1
(this runs the Bonnie benchmark - it may take a little while)
./lmp1 -test suite1
(run Test Suite 1 - this has to work for stage1)
make test
(run all tests - this has to work for stage2)
```

2. Read through the provided .c and .h files and understand how this project is organized:

```
bonnie.c - a version of the filesystem benchmark
fileio.c - file I/O functions to be implemented
fileio.h - declaration of file I/O functions
restart.c - restart library (available for use in fileio.c)
restart.h - declaration of restart library functions
util.c - useful utility functions
util.h - declaration of useful utility functions and macros
```

In particular, pay close attention to the comments in fileio.h and bonnieb.c. You should understand what each of the following functions in bonnie.c does before undertaking the remainder of the MP:

```
fill_file_char()
file_read_rewrite()
file_read_rewrite_block()
fill_file_block()
fill_read_getc()
file_read_chunk()
newfile()
```

Step 2: Basic I/O operations

Implement file_read, file_write and file_info operations in fileio.c.

If done correctly, your code should pass all suite1 tests:

```
./lmp1 -test suite1
Running tests...
 1.read                ::pass
 2.info                ::pass
 3.write               ::pass

Test Results:3 tests,3 passed,0 failed.
```

IMPORTANT: fileio.c is the only file you should modify for this step.

Step 3: More filesystem operations

Implement file and directory operations for suite2 (dir_create and dir_list), suite3 (file_create and file_remove), and suite4 (file_checksum and dir_checksum).

You can test each operation and test suite individually:

```
./lmp1 -test dirlist
./lmp1 -test suite2
```

All tests should now pass:

```
make test
Running tests...
 1.read                ::pass
 2.info                ::pass
 3.write               ::pass
 4.dirlist             ::pass
 5.dircreate           ::pass
 6.remove              ::pass
 7.create              ::pass
 8.filechecksum        ::pass
 9.dircheckSum         ::pass

Test Results:9 tests,9 passed,0 failed.
```

Step 4: Performance testing

In this step, we will change parts of Bonnie to use our filesystem interface.

Make the function file_read_rewrite_block() in bonnie.c to call your fileio.c functions instead of POSIX I/O operations. When answering the questions below, use this modified version of bonnie.c.

Before making this change, it's a good idea to write pseudocode comments for what each part of file_read_rewrite_block() does, so that you understand the code and can perform the change correctly. There may not be an exact one-to-one correspondence between our filesystem interface and the POSIX commands.

Note: In future LMPs, we will be using the fileio.h interface in a similar manner, but we will call the functions remotely, via a message queue.

Questions

Q1. Briefly explain what the following code from bonnie.c does:
`if ((words = read(fd, (char *) buf, Chunk)) == -1) ...`

Q2. Is the above an example of a block read or a character read? What is the value of the variable 'words' if the read succeeds? Fails?

Q3. Explain the meaning of the flag value (O_CREAT | O_WRONLY | O_APPEND) for the POSIX function open().

Q4. Run Bonnie. What is being measured by each test function?

Q5. Look at the summary results from the Bonnie run in Q4. Does Bonnie measure latency, throughput or something else? Justify your answer.

Q6. Compare character reads with block reads using Bonnie. Which is faster? Why do you think this is the case?

Q7. Copy and paste the performance measures output when running Bonnie benchmarks in a local directory and again in an NFS-mounted directory. Is one kind of disk access noticeably slower over the network, or are all tests significantly slower?

Your home directory may be an NFS mount, whereas /tmp and /scratch are local disks. To test your code in /tmp, do the following:

```
mkdir /tmp/your_username
cp lmp1 /tmp/your_username
cd /tmp/your_username
./lmp1
(record the output)
cd
rm -fr /tmp/your_username
```

Q8. How does Bonnie handle incomplete reads, e.g., due to interruptions from signals? Justify why Bonnie's approach is good or bad for a filesystem benchmark program.

Q9. By now you should be very familiar with the self-evaluation test harness we provide for the MPs. Examine the function test_file_read() in lmp1_tests.c, which tests your file_read() function from Step 2.

What does this test check for, specifically? You may want to copy and paste the code for this function in your answer, and annotate each quit_if or group of related quit_ifs with a comment.

Submit all code for this part of the assignment and answers to questions within you answers.txt file.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!

Part 2 – Memory Management

This machine problem will focus on memory. You will implement your own version of malloc() and free(), using a variety of allocation strategies.

You will be implementing a memory manager for a block of memory. You will implement routines for allocating and deallocating memory, and keeping track of what memory is in use. You will implement four strategies for selecting in which block to place a new requested memory black:

1) First-fit: select the first suitable block with smallest address.
2) Best-fit: select the smallest suitable block.
3) Worst-fit: select the largest suitable block.
4) Next-fit: select the first suitable block after the last block allocated (with wraparound from end to beginning).

Here, "suitable" means "free, and large enough to fit the new data".

Here are the functions you will need to implement:

initmem():
  Initialize memory structures.

mymalloc():
  Like malloc(), this allocates a new block of memory.

myfree():
  Like free(), this deallocates a block of memory.

mem_holes():
  How many free blocks are in memory?

mem_allocated():
  How much memory is currently allocated?

mem_free():
  How much memory is NOT allocated?

mem_largest_free():
  How large is the largest free block?

mem_small_free():
  How many small unallocated blocks are currently in memory?

mem_is_alloc():
  Is a particular byte allocated or not?

We have given you a structure to use to implement these functions. It is a doubly-linked list of blocks in memory (both allocated and free blocks). Every

malloc and free can create new blocks, or combine existing blocks. You may modify this structure, or even use a different one entirely. However, do not change function prototypes or files other than mymem.c.

IMPORTANT NOTE: Regardless of how you implement memory management, make sure that there are no adjacent free blocks. Any such blocks should be merged into one large block.

We have also given you a few functions to help you monitor what happens when you call your functions. Most important is the try_mymem() function. If you run your code with "mem -try <args>", it will call this function, which you can use to demonstrate the effects of your memory operations. These functions have no effect on test code, so use them to your advantage.

Running your code:

After running "make", run

1) "mem" to see the available tests and strategies.
2) "mem -test <test> <strategy>" to test your code with our tests.
3) "mem -try <args>" to run your code with your own tests
   (the try_mymem function).

You can also use "make test" and "make stage1-test" for testing. "make stage1-test" only runs the tests relevant to stage 1.

As in previous MPs, running "mem -test -f0 ..." will allow tests to run even after previous tests have failed. Similarly, using "all" for a test or strategy name runs all of the tests or strategies. Note that if "all" is selected as the strategy, the 4 tests are shown as one.

One of the tests, "stress", runs an assortment of randomized tests on each strategy. The results of the tests are placed in "tests.out" . You may want to view this file to see the relative performance of each strategy.

Stage 1

Implement all the above functions, for the first-fit strategy. Use "mem -test all first" to test your implementation.

Stage 2

A) Implement the other three strategies: worst-fit, best-fit, and next-fit. The strategy is passed to initmem(), and stored in the global variable "myStrategy".

Some of your functions will need to check this variable to implement the correct strategy.

You can test your code with "mem -test all worst", etc., or test all 4 together with "mem -test all all". The latter command does not test the strategies separately; your code passes the test only if all four strategies pass.

Questions
=========

1) Why is it so important that adjacent free blocks not be left as such? What would happen if they were permitted?

2) Which function(s) need to be concerned about adjacent free blocks?

3) Name one advantage of each strategy.

4) Run the stress test on all strategies, and look at the results (tests.out). What is the significance of "Average largest free block"? Which strategy generally has the best performance in this metric? Why do you think this is?

5) In the stress test results (see Question 4), what is the significance of "Average number of small blocks"? Which strategy generally has the best performance in this metric? Why do you think this is?

6) Eventually, the many mallocs and frees produces many small blocks scattered across the memory pool. There may be enough space to allocate a new block, but not in one place. It is possible to compact the memory, so all the free blocks are moved to one large free block. How would you implement this in the system you have built?

7) If you did implement memory compaction, what changes would you need to make in how such a system is invoked (i.e. from a user's perspective)?

8) How would you use the system you have built to implement realloc? (Brief explanation; no code)

9) Which function(s) need to know which strategy is being used? Briefly explain why this/these and not others.

10) Give one advantage of implementing memory management using a linked list over a bit array, where every bit tells whether its corresponding byte is allocated.

Submit all code for this part of the assignment and answers to questions within you answers.txt file.

Reminder: Do not copy or plagiarize any code from any other student in the course and be sure to cite all online references.

Do not copy or plagiarize from any source online. Any student found doing so will receive a 0 for the assignment portion of the course. My goal is to maximize your learning, so please focus on that!