Mini Project 3: Memory management and File Systems

Due: April 8, 2020, 11:59pm

This project has two parts, one on memory management and one on file systems. All parts must be written in C or C++.

Part I: Memory management - paging

This purpose of this component of the mini project is to implement a portion of the virtual to physical address translation in a pure paging based scheme. You goal is to write a program that takes a sequence of virtual addresses and extract the page number and offset for each address using bit-wise operations.

Assume that virtual addresses are unsigned integers (i.e., *unsigned int*) and that we only look at 16 bits of the integer representing a virtual address. Thus, even if your machine represents integers as 32 or 64 bits, we are only concerned with the lower 16 bits of an integer to represent the virtual address. Your program should take the following inputs from an input file:

```
n   /* the n lowest significant bits that represent the offset */
m   /* the next m bits that represent the page number; assume that n+m is
always 16 */
v1   /* first virtual address that needs to be mapped to a page number and
offset */
v2   /* second virtual address */
...  /* a sequence of virtual addresses, one per line until the end of the
file */
```

Your program should read the above input file, and in particular, read each virtual address as an unsigned integer and compute and print out the page number and offset for each address. Assume that the virtual addresses are correct and no error checks are needed. To compute the offset, use bit operations in C/C++ to extract the n least significant bits from v and print out the resulting integer. To compute the page number, extract the next m significant bits from v and print out the integer representing these bits.

You can check the result of your program by hand-computing the page number and offset as **v DIV 2^n** and **v MOD 2^n** However your program should not use division and mod operations to compute these values and should use bit-wise operations instead (just as a hardware implementation of paging would use).

The output of your program should look like:

```
virtual address v1 is in page number p and offset d
virtual address v2 is in page number p and offset d
...
```

Here is a tutorial on bit operations in C and C++: tutorial. Pay particular attention to the right shift operator as well as the using bit-masks and the bit-wise AND operation, which are concepts that will be helpful for this mini project.

Part II: File Systems

The goal of part 2 of mini project is to write a simple UNIX-like file system based on the topics covered in class. The file system you will write makes the following simplifying assumptions:

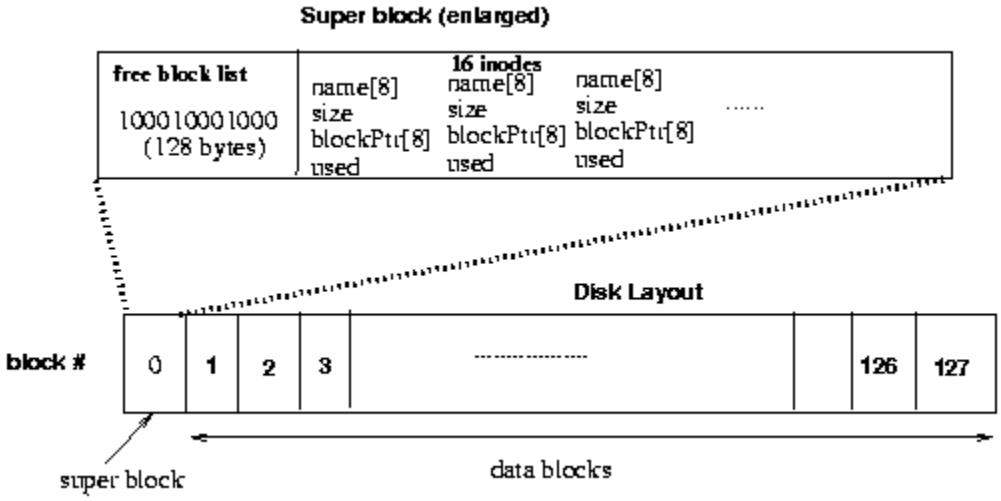
- The file system resides on a disk that is 128KB in size.
- There is only one root directory. No subdirectories are allowed.
- The file system supports a maximum of 16 files.
- The maximum size of a file is 8 blocks; each block is 1KB in size.
- Each file has a unique name, the file name can be no longer than 8 characters.

The layout of your 128 KB disk is as follows

- The first 1KB block is called the super block. It stores the free block list and index nodes (inode) for each file.
- The remaining 127 1KB blocks store data blocks of the files on your file system.
- The exact structure of the super block is as follows.
    - The first 128 bytes stores the free block list. Each entry in this list indicates whether the corresponding block is free or in use (if the i-th byte is 0, it indicates that the block is free, else it is in use). Intially all blocks except the super block are free.
    - Immediately following the free block list are the 16 index nodes, one for each file that is allowed in the file system. Initially, all inode are free. Each inode stores the following information:
    - `char name[8]; //file name`
    - `int32 size;      // file size (in number of blocks)`
    - `int32 blockPointers[8]; // direct block pointers`
    - `int32 used;             // 0 => inode is free; 1 => in use`

      Note that each inode is 48 bytes in size. Since you have 16 of these, the total size occupied by the inodes is 768 bytes. The free/used block

information (mentioned above) is 128 bytes. So the total space used in the super block is 896 bytes, leaving the last 128 bytes empty.



You need to implement the following operations for your file system.

- *create(char name[8], int32 size)*: create a new file with this name and with these many blocks. (We shall assume that the file size is specified at file creation time and the file does not grow or shrink from this point on)
- *delete(char name[8])*: delete the file with this name
- *read(char name[8], int32 blockNum, char buf[1024])*: read the specified block from this file into the specified buffer; blockNum can range from 0 to 7.
- *write(char name[8], int32 blockNum, char buf[1024])*: write the data in the buffer to the specified block in this file.
- *ls(void)*: list the names of all files in the file system and their sizes.

Getting Started for Part II

- We will use a 128KB file to act as the "disk" for your file system. I have provided a program to create this file for you. Run this program by typing the command

  ```
  create_fs disk0
  ```

  This will create a file with the name

  ```
  disk0
  ```

in your current directory. The program also "formats" your file system---this is done by initializing all blocks in the super block to be free and marking all 16 inodes to be free.

If you work on this assignment on your home PC, you can find the source code for the create_fs program [C version .]

- Here is template code for your file system.
- Remember that your file system must be persistent. If you shutdown your program and then restart it at a later time, all files on your file system must be intact.
- Your file system must read and write to disk only in multiples of the 1KB block size.
- **Input file**

  You program should take input from a input file and perform actions specified in the file, while printing out the result of each action. The format of the input file is as follows.

```
diskName  // name of the file that emulates the disk
C fileName Size //create a file of this size
D fileNAme     // delete this file
L              // list all files on disk and their sizes
R fileName blockNum // read this block from this file
W fileName blockNum // write to this block in the file (use a dummy
1KB buffer)
```

  An sample input file looks like this:

```
disk0
C file1 3
W file1 0
W file1 1
C mini-project.java 7
L
C file2 4
R file1 1
D mini-project.java
L
```

  A sample input file is available. Be sure to print out what your program does after reading each line from this file. It'd also be helpful if you printed out the disk addresses of any block that you allocate, deallocate, read or write.

How to Turn in Mini Project 3

All of the following files must be submitted on Canvas as a zip file to get full credit for this assignment.

1. Your zip file should contain a copy of all source files.
2. Part 1 and part 2 should be in two separate directories in your zip file.
3. Your zip file should contain a copy of a README file for each part. The README file should identify your mini project partner and contain an outline of what you did for the assignment. It should also explain and motivate your design choices. Keep it short and to the point, while explaining explain your design decisions, data structures and algorithms.
   If your implementation does not work, you should also document the problems in the README, preferably with your explanation of why it does not work and how you would solve it if you had more time. **Of course, you should also comment your code. We can't give you credit for something we don't understand!**
4. Your zip file should contain a copy showing sample output from your programs. Feel free to include sample input files you have used that are different from the ones we have provided
5. Your zip file should contain build instructions stating exactly how to compile your code on the Linux machines for the C or C++ implementations. Preferably, these will be in the form of Makefiles. Here is a Makefile for this C source. This can be run by the program make. Alternatively, you could write the instructions in your README file, but we should be able to copy the commands from your README and paste it into a terminal to compile your source code.
6. Your README file should contain run instructions stating exactly how to execute your compiled code on the Linux machines for C or C++ implementations. These instructions should include a description of any command line arguments your program expects or accepts. If your program prints a usage message, then it is sufficient to say how to get your program to print that message.
7. **Important:**All files for the first and second parts of this mini project should be distinct from one another. Feel free to keep them in separate sub-directories. This will allow us to grade each part separately.
8. Individual Group Assessment (for students working in groups) – Look out for iPeer evaluations.
9. **Note:** We will *strictly* enforce policies on cheating. Remember that we routinely run similarity checking programs on your solutions to detect cheating. Please make sure you turn in your own work.

You should be very careful about using code snippets you find on the Internet. In general your code should be your own. It is OK to read tutorials on the web and use these concepts in your assignment. Blind use of code from web is strictly **disallowed**. Feel free to check with us if you have questions on this policy. And be sure to document any Internet sources/ tutorials you have used to complete the assignment in your README file.

10. **Late Policy:** Please refer to the course syllabus for late policy on mini project assignments. This late policy will be strictly enforced. Please start early so that you can submit the assignment on time.