Assignment 05

100 points

You must work on you own

Due: **beginning of the class**, **May 3rd**

6 questions – see each question for points allocated for it

Make sure to include the **single examinee affidavit** at the beginning of your answer sheet. You must work on your own for this assignment, it would be a red flag if we find submissions from two students are with exact same incorrect answers in multiple parts.

For all pseudocode, magic numbers are fine as long as proper comments are there to show what they mean.

1. (**20** points) Write pseudocode to support hard link creation and removal for a file system with indexed allocation.
   - **createHardlink(srcPath, linkPath)** - Create a hard link to an existing file path
     - srcPath could be a path to a file or a directory. (Hint: can you create a hard link to a directory?)
   - **removeHardlink(linkPath)**

   Hard links of a same file point to the root inode of the file. You may suppose a field in the i-node metadata that would let the operating system know when a file should be deleted. This field needs to be tracked and updated in the hardlink operations.

   Error checking is not required. And assume the following file system functions are provided:
   - getInodeBlockNum(filePath) – Given a file path, return the block number of its root i-node.
   - readDatablock(blockNum) – Read data from the specified block. Assume data returned for an inode block number can be used as an inode object (i.e., an instance of an i-node data structure).
   - writeDatablock(blockNum, blockData) – Write block data to specified block in the file system.
   - getDir (filePath) – Given a file path, return the containing directory of a file path or the current directory if only a filename is given.
   - getfileName(filePath) – Return the filename portion of a file path.
   - addFileToDir(dir, filename, inodeBlk) – Add filename to the specified directory with a root i-node block of inodeBlk.

- removeFileFromDir(dir, filename) – Remove filename from the specified directory.
- releaseInode (inodeBlk) – Given the block number of the root i-node of a file, reclaim all i-nodes associated with this file to the free i-node pool and all data blocks to the free data block pool.

2. (**20** points) An i-node file system supports 10 direct blocks, 3 single indirect blocks, and 2 double indirect blocks. Assuming 4KB block size, each i-node pointer entry uses 8 bytes.

   a) What is the biggest file size supported by the system?
   b) Write pseudocode to implement the following function:

   ```
   // Given block number of the root inode of a file and a byte offset (Nth) into the
   // file, return the block number of the data block containing the Nth byte.
   BlockNum findBlockOfOffset(BlockNum rootInodeBlk, unsigned long nthByte)
   ```

   Error checking is not required. And **assume**:

   - A readDatablock function as in question 1.
   - Each inode has to fit into one data block.
   - The root inode structure has 3 arrays containing the block numbers for the direct, indirect, and double indirect blocks:

     ```
     #define NUMOFDIRECT 10
     #define NUMOFSINGLEINDIRECT 3
     #define NUMOFDOUBLEINDIRECT 2
     BlockNum directBlocks[NUMOFDIRECT];
     BlockNum singleIndirect[NUMOFSINGLEINDIRECT];
     BlockNum doubleIndirect[NUMOFDOUBLEINDIRECT];
     ```

   - Each indirect i-node structure (Single and double indirect nodes) has an array of block numbers as:

     ```
     // Hint: NUMBLOCKS number of blocks (pointers) in each single or double
     // indirect inode can be derived from block size and assumption b (see
     // above)
     BlockNum blockNumber[NUMBLOCKS];
     ```

3. (**10** points) Is the following disk operation idempotent? Replacing every white space with an asterisk at the end of each line in a file. Justify your answer.

4. (**15** points) Suppose a bitmap is used for tracking a disk block free list. The bitmap is represented as an array of 32-bit integers (i.e., each word is an 32 bit integer). Write **C syntax pseudocode** to implement the following function:

```
/*
 * Given a bitmap (first argument) stored in an array of words, starting from the
 * beginning of the bitmap, find the first run of consecutive free blocks (a hole)
 * whose size has at least the number of needed blocks (second argument), and
 * return the starting block index of the found run of free blocks. If a big enough
 * hole  cannot be found to fit the number of needed blocks, return -1.
 */
```

#define BITSPERWORD 32

int findFreeblocks (int words[], int numOfNeededBlocks)

Error checking is not required. And assume:

1) **int** type has 32 bits.
2) Each bit in the bitmap represents one block, value 1 is occupied, 0 is free.
3) The block index starts at 0 from the first bit to the left (most significant) of the first word, incrementing one at a time to the right, then continuing to the next word. E.g., the block index of the first bit of the second word would be 32, etc.
4) **NO** need to worry about the endianness of storing each integer / word.

Hint:

1) To extract each bit from the word, use a bit mask and bitwise and. (recall a3 program)
2) The hole (run of free blocks) would start from a bit with 0 in a word entry and runs until it reaches a bit with 1. It could go across the word boundaries.

5. (**15** points) A toy magnetic disk has two tracks with 10 blocks each. Inner track has blocks 0 - 9 and outer track has blocks 10 - 19. Disk spins counterclockwise at a speed of 100 rotations per second. Suppose it takes 2.4ms for the disk head to travel from track-to-track. Draw a disk and label the blocks such that there is the **minimal** disk skew that allows us to read block 10 after block 9 is read with minimal delay. Note: if a block (to be read) passes (with the spin) the disk head before the disk head moves to the outer track. Disk head needs to wait until the block spins back under its position before it can proceed to read.

6. (**20** points) A hardware supports a single timer device. Write the **pseudocode** (see below) to provide support of alarm services to user processes who request the use of alarms.
   You must support multiple alarms using the single timer device. To accomplish that, you would use a data structure to remember a list of (timestamp, pid) entries that tracks the alarm times and the processes that requested the alarms (Refer to I/O slides 3 – 6).

   You would use memory mapped I/O to set the alarm time for the nearest alarm requested by a process and enable the alarm. The timer always counts toward the nearest future alarm time / deadline.
   - An alarm is set on this hardware by writing the alarm deadline timestamp to memory location **0x12B0**.
   - To enable the alarm after setting a deadline, set **bit 7** of memory location **0x12B8**.

   When the current alarm deadline is reached, it sets off alarm interrupt and signals the corresponding request process.

   Error checking is not required.

```
/*
 * This interface is called by a system call to add an alarm request for calling process
 * (processId) at a specified time. Assume access to any data structures you may need.
 * (real interface would be more complicated with register or stack arguments
 * specified)
 * Hint: As mentioned above, the timer would always enable and count towards the nearest
 * future alarm time.
 */
```

bool addAlarm(alarmTimestamp, processId)

```
/*
 * Interrupt service routine (ISR) that is called when the hardware timer reaches the current
 * alarm time / deadline, i.e., the current alarm was set off.
 *
 * Sets the next nearest alarm time (if any).
 *
 * Sends a SIGALRM to the process that requested the alarm that was set off using function
 *    sendsignal(sig, processId)
 */
```

void setoffAlarm()