CS 111 Midterm Examination #2
Spring Quarter, 2021
Solutions

You have 90 minutes for this examination; the number of points for each question indicates roughly how many minutes you should spend on that question. While taking this exam, you may consult any materials available on your personal computer at the time of the exam, as well as materials on the class Web site; you may not search the Internet during the exam. If there is a trivial detail that you need for one of your answers but cannot recall, you may ask the course staff. Fill in the fields of this PDF file, then submit it at www.gradescope.com.

By filing this examination electronically, you acknowledge and accept the Stanford University Honor Code, and you affirm that you have neither given nor received aid in answering the questions on this examination.

_____
(Name)

_____
(SUNet email id)

Problem	#1	#2	#3	#4	#5	#6	Total
Score							
Max	10	3	10	10	10	35	78

Problem 1 (10 points) Friendly advice

(a) (3 points) A friend comes to you with a suggestion for disk management. Your friend points out that modern operating systems don't allow disk space to become totally full, and suggests that the simplest way to implement this is to permanently block off a range of blocks at one end of the disks, so those blocks are never allocated. Is this a good idea or a bad idea, and why?

Answer: this is a bad idea. The reason for preventing the disk from filling is so that there are lots of free blocks available for allocation. Ideally they are scattered across the disk, so that, when extending a file, the next block can be allocated close to the previous block to minimize seeks. If the unused blocks are cordoned off at one end then (a) they can't actually be allocated for files, which defeats the purpose, and (b) even if they could be allocated, they probably wouldn't help in reducing seeks.

(b) (3 points) Your friend also suggests a performance improvement for the clock algorithm. Rather than skipping over pages whose reference bit is set, your friend suggests just evicting the first page the algorithm encounters. Is this a good idea or a bad idea, and why?

Answer: this is also a bad idea. This would result in FIFO replacement, whereas the normal clock algorithm provides an approximation to LRU.

(c) (4 points) Your friend then suggests that it would be better for a file system to eliminate inodes as a separate structure and simply store all of the inode information for a file directly in the directory entry for the file. Give one advantage and one disadvantage of this new approach.

Advantage: fewer distinct disk blocks would need to be accessed when opening files, which could result in better performance.

Disadvantage: hard links would be difficult or impossible to implement (it would require duplicating inode information each of the directory entries and maintaining consistency between these copies).

Problem 2 (3 points)

Internal fragmentation occurs both in virtual memory systems based on paging and in file systems. In which of these two subsystems is internal fragmentation a more significant issue, and why?

Answer: internal fragmentation is considerably worse in file systems, because most files are small in comparison to the block size. Thus, wasting 1/2 block per file (on average) is a large penalty. For virtual memory systems, segments such as code and data tend to be much larger than the page size, so wasting 1/2 page for each segment only wastes a small fraction of memory.

Problem 3 (10 points)

Consider the segmented approach to memory management discussed in class, and assume that the segment number is taken from the top few bits of the virtual address. That mechanism allows segments to grow upwards (to larger virtual addresses), but not downwards. For example, the mechanism would not allow a stack segment to initially occupy the top virtual addresses of a segment's region and grow downward as needed. Describe how the mechanism could be modified to support segments that grow downwards as well as upwards, while retaining a segmented approach. Note: in order for your mechanism to be implemented efficiently in hardware, it needs to use only simple operations such as addition, subtraction, comparison, and bit concatenation.

Answer: one simple approach is to have two bounds in each entry in the segment map. The first is an upper bound, similar to what was described in class, and the second is a lower bound. During address translation, if the segment offset is either greater than the upper bound or less than the lower bound, then a segmentation fault occurs.

Problem 4 (10 points)

This question concerns the 4.3 BSD multi-level file structure which uses zero, one, or two levels of indirection, depending on the file size. Assume that blocks are 4 Kbytes, block pointers are 32 bits, and the file size stored in inodes is 64 bits.

(a) (4 points) What is the largest file that can be supported by this structure? You may round your answer to the nearest power of 2. You may use a calculator to compute the answer, but be sure to show enough work for us to understand how you arrived at your answer.

Answer: 2^{32} bytes. Each indirect block holds (4096 bytes per block) / (4 bytes per pointer), which is 1024 (2^{10}) pointers. Two levels of indirection allows $2^{10}\times2^{10}$ total data blocks, and each data block holds 2^{12} bytes, so the total file size is $2^{10}\times2^{10}\times2^{12}$ bytes, or 2^{32} bytes. This is just an approximation, since it doesn't include the direct and singly-indirect pointers.

(b) (6 points) The maximum file size could be increased by adding more block pointers to the inode, for 3-level indirection, 4-level indirection, and so on. However, there will be a point where it doesn't help to add more levels of indirection. What is the largest number of levels of indirection that makes sense, and why?

Answer: 4 levels. With 3 levels of indirection, the maximum file size would increase to about 2^{30} blocks, or 2^{42} bytes. With 4 levels, files could have up to about 2^{40} blocks, or 2^{52} bytes. However, block pointers are only 32 bits, so there wouldn't be enough blocks in the disk to fully populate 4 levels of indirection. Additional levels of indirection wouldn't provide any additional benefit unless the size of block pointers were increased.

Problem 5 (10 points)

Your friend from Problem 1 has built a file system that uses write-ahead logging for crash recovery; it logs both metadata and file data. However, you notice that the system does not seem to recover properly after some crashes. In looking over the file system code, you discover that in some situations the system creates a log record of the form "append data D to the file with i-number I" (the log record contains an opcode indicating an append operation, plus the new data and the file's i-number).

(a) (5 points) How can this log record cause incorrect behavior after a crash?

Answer: the problem is that the append operation isn't idempotent: if the operation is done multiple times, it will produce a different result than if it is done only once. This is a problem because it's possible that the operation has already been reflected on disk at the time of a crash, so replaying the log may perform the operation a second time. In general we don't know whether operations have already been performed or not when a crash occurs, so it's important that all operations in the log are idempotent.

(b) (5 points) Describe how the log record could be restructured to eliminate this problem. Be precise about exactly what information should be present in the log record, and why that fixes the problem.

Answer: the solution is to change the log operation so that it is idempotent. One possibility is to change the operation from an append to a write: "write data D at offset O in the file with i-number I".

Problem 6 (35 points)

Modify your solution to Project 7 ("Unix V6 File System") to support symbolic links when opening files.

(a) (5 points) What changes would need to be made to existing structure declarations in the project code to support symbolic links?

Answer: there needs to be a new inode field indicating that a particular file is a symbolic link. Ideally this would use the existing `IFMT` field, but unfortunately all of those combinations are already in use and there isn't room in `i_mode` to expand `IFMT`. The solution here assumes the addition of a new field in `struct inode` declared as follows:

```
uint8_t i_symlink;
```

If this value is nonzero, it overrides the `IFMT` bits and indicates that the file is a symbolic link. In this case, the first block of the file contains a null-terminated string containing the target of the symbolic link.

In addition, there needs to be a new `#define` named `MAX_SYMLINKS`, which sets an upper bound on the number of symbolic links that can be traversed in a particular path lookup (this prevents infinite loops).

(b) (10 points) Modify the `pathname_lookup` function you wrote for Project 7 to properly handle symbolic links. You do not need to write the code to create symbolic links, but your answer to (a) must make it clear how those links will appear in the file system data structures. You may assume that the contents of a symbolic link are never longer than 512 bytes, and that the target of a symbolic link is always in the same `struct unixfilesystem` as the source.

Include here or on the following pages any new functions or structure declarations that you write, as well as any existing ones that you modify.

Answer: see next page.

Additional working space for Problem 6:

```
int pathname_lookup(struct unixfilesystem *fs, const char *pathname) {
    char clone[strlen(pathname) + 1];
    strcpy(clone, pathname);
    if (clone[0] != '/')
        return -1;
    return lookup2(fs, ROOT_INUMBER, clone+1, MAX_SYMLINKS);
}

/* The arguments to lookup2 have the same meaning as for pathname_lookup,
 * except that path refers to a copy of the path that this function can
 * overwrite. Symlinks_left is a count of how many more symlinks can be
 * traversed before aborting. */
int lookup2(struct unixfilesystem *fs, int inumber, char *path, symlinks_left)
{
    if (path[0] == '\0')
        return inumber;
    while (path != NULL) {
        const char *component = strsep(&path, "/");
        struct direntv6 dir_entry;
        if (directory_findname(fs, component, inumber, &dir_entry) == -1)
            return -1;
        inumber = dir_entry.d_inumber;

        struct inode in;
        if (!inode_iget(fs, inumber, &in))
            return -1;
        if (in.i_symlink) {
            char buf[DISKIMG_SECTOR_SIZE];
            if (symlinks_left == 0)
                return -1;
            if (file_get_block(fs, inumber, 0, &buf) != DISKIMG_SECTOR_SIZE)
                return -1;
            if (buf[0] == ‘/’)
                inumber = lookup2(fs, ROOT_INUMBER, buf+1, symlinks_left-1);
            else
                inumber = lookup2(fs, inumber, buf, symlinks_left-1);
            if (inumber < 0)
                return inumber;
        }
    }

    return inumber;
}
```

Additional working space for any problem, if needed