Exam 3

601.418/618 Operating Systems

May 9, 2024

Complete all questions.

Time: 90 minutes.

I affirm that I have completed this exam without unauthorized assistance from any person, materials, or device.

Signed: _____

Print name: _____

Date: _____

Question 1. [5 points] In the Unix Version 6 filesystem, addresses of disk blocks are 16 bits in size. Assuming that disk blocks are $512 = 2^9$ bytes, what is the size in bytes of the largest hard disk that could be completely utilized by a Unix Version 6 filesystem? Show your work.

Question 2. [10 points] A Unix V6 inode has an array of 8 disk block addresses which specify the allocated blocks for the file or directory represented by the inode. Assume that disk blocks are $512 = 2^9$ bytes.

(a) For a "small mode" file or directory, each of the 8 addresses in the array is a direct pointer to a storage block. What is maximum size in bytes of a "small mode" file or directory? You may express your answer as a formula. Show your work.

(b) For a "large mode" file or directory, the first 7 addresses are pointers to singly-indirect blocks, and the last address is a pointer to a doubly-indirect block. What is the maximum size in bytes of a "large mode" file or directory. Recall from Question 1 that disk addresses are 16 bits (i.e., 2 bytes.) You may express your answer as a formula. Show your work.

Question 3. [10 points] Unix filesystems have traditionally supported "sparse" files, in which a process can use the `lseek` system call to set the current file position beyond the current end of the file, and any bytes in the "hole" created between the original end of file offset and the new end of file offset are returned as zeroes if read.

Briefly explain why and how the OS can avoid allocating storage space for a hole created in this manner. Your answer should be specific about what the inode and associated metadata will look like for a file with a hole. Drawing a diagram is recommended.

Question 4. [10 points]

(a) What information is stored in a Unix directory entry?

(b) In the Unix filesystem, the storage for a directory's contents are managed in the same way as the storage for a file's contents. Does that mean that it would make sense to allow a process to use the `write` system call to modify a directory's contents, assuming that it has permission to modify the directory? Briefly explain why or why not.

Question 5. [15 points] Assume that a filesystem implementation uses an array of `uint32_t` elements as a bitmap to keep track of which disk blocks are in use and which are available for allocation. If the filesystem has N storage blocks, for a storage block with address i ($0 \leq i < N$), if bit i in the bitmap is set to 0, block i is available for allocation, and if bit i of the bitmap is set to 1, then block i is in use.

The following functions show how getting and setting a bit could be implemented:

```
// set bit i to 1 or 0
void set_bit(uint32_t *bitmap, unsigned N, unsigned i, int val) {
  assert(i < N);
  assert(val == 0 || val == 1);
  if (val == 1)
    bitmap[i / 32] |= (1 << (i % 32));   // set bit
  else
    bitmap[i / 32] &= ~(1 << (i % 32));  // clear bit
}

// get the value of bit i (1 or 0)
int get_bit(uint32_t *bitmap, unsigned N, unsigned i) {
  assert(i < N);
  return (bitmap[i / 32] & (1 << (i % 32))) != 0;
}
```

On the next page, complete the implementation of the `reserve` function. It should

1. Find a series of `nblocks` consecutive available storage blocks
2. Mark each block in the series as allocated (by setting their bits to 1)
3. Set the variable that pointed-to by `start` to the address of the first block in the series

If these steps are successful, the function should return 1, otherwise it should return 0 (if no sufficiently-large series of consecutive available blocks exists.)

Your code may call the `set_bit` and `get_bit` functions shown above.

[Question 5 continues on the next page.]

[Question 5 continues.]

```
int reserve(uint32_t *bitmap, unsigned N, unsigned nblocks, unsigned *start) {
```

Question 6. [10 points] The Unix/Linux `fsync` system call has the following signature:

```
int fsync(int fd);
```

When a process calls `fsync`, the kernel flushes the modified data and/or metadata of the file indicated by `fd` to persistent storage (disk or SSD.)

Briefly sketch what the implementation of `fsync` in the OS kernel would look like. You can use pseudo code, but be explicit about what data structures (both on-disk and in-memory) are accessed.

Question 7. [10 points] Consider the following program (assume appropriate headers have been included):

```
void error(void) {
  fprintf(stderr, "error: %s\n", strerror(errno));
  exit(1);
}

int main(void) {
  int fd;
  if ( (fd = open("/home/daveho/out.txt", O_WRONLY|O_CREAT, 0600)) < 0
       || write(fd, "hello, world\n", 13) != 13
       || fsync(fd) != 0 )
    error();
  return 0;
}
```

Describe a scenario where the call to `fsync()` in this program succeeds (returns 0), but at a later time, the command `cat /home/daveho/out.txt` fails to print the output `hello, world`. Hint: if `fsync()` succeeds, the indicated file's data and metadata has been written to persistent storage. Can you think of a reason why this would not guarantee the success of the `cat` command? Assume that the user running the `cat` is the same user that ran the above program. Note that the octal file permissions constant `0600` allows read and write permission for the user.

Question 8. [5 points] In the FAT16 file system, each entry in the FAT (File Allocation Table) represents one disk storage block, and sequence of storage blocks allocated to a specific file are tracked by having each FAT entry representing a storage block allocated to the file contain the index of the next allocated block, with the exception of the FAT entry for the last storage block allocated to a file, which contains a special terminator value. The FAT entries are 16 bits (2 bytes) in size, allowing for approximately 2^{16} distinct storage blocks maximum.

Assume that a process is sequentially reading the data in a file on a FAT16 filesystem with N total storage blocks. How many disk seeks will the OS kernel need to perform in order to determine the sequence of storage blocks storing the file's data? Note that this total should *not* include seeks needed to read the file's actual data. Assume that the OS kernel will use appropriately techniques to minimize seeks. If any data structures are involved in optimizing access to filesystem metadata, describe those data structures.

Question 9. [5 points] Briefly describe the problem that journaling filesystems are meant to address.

Question 10. [5 points] Compared to a "normal" filesystem (e.g., the original Unix filesystem or the Unix Fast Filesystem), what is the most significant performance cost associated with journaling filesystems?

Question 11. [5 points] The filesystem buffer cache is used for both *caching* of filesystem data and *buffering* of I/O operations. Briefly explain how the buffer cache achieves both caching and buffering, and how each benefits system performance.

Question 12. [10 points] The following function is meant to rename a regular file (not a directory) from its original name to a new name:

```
// returns true if successful, false if not
bool rename(const char *orig_name, const char *new_name) {
  return link(orig_name, new_name) == 0 &&
         unlink(orig_name) == 0;
}
```

Note that both `link` and `unlink` return 0 to indicate success. The `link` system call creates a new hard link to a file, and the `unlink` system call removes a hard link to a file.

(a) Describe a scenario where a call to `rename` succeeds, but a system crash would result in the file being available as *neither* the old name or the new name. Assume that the filesystem is not a journaling filesystem.

(b) If scenario (a) occurs on a typical Unix or Linux system, describe how the contents of the file could be recovered.

[Extra page for answers and/or scratch work.]

[Extra page for answers and/or scratch work.]