PA5

HOJOON KIM

December 2024

1 New data structures

1.1 Hash Table

```c
#define HASH_TABLE_SIZE 12809
static int hash=0;
struct hash_entry {
    char name[MAXPATH]; //   32
    uint inum;     // inode
    uint off;
};

struct hash_table {
  struct hash_entry lines[HASH_TABLE_SIZE];
  struct spinlock lock;
};
struct hash_table table;

```

Currently, dirlookup() operates by iterating through the entire datablock to retrieve the inum from the block containing the specified name. I believe this method results in the largest overhead. To reduce the retrieval time, I implemented a hashtable. The hashtable adds new entry when dirlink() is called to insert a new entry into the directory and removes entry from the table when unlink() is called to delete an entry.

In the hashtable implementation, the name is used as the input to the hash function, allowing us to directly map to the corresponding block and retrieve the inum. The key feature of the hashtable is ensuring that different name values produce distinct index values, enabling constant time O(1) lookups. This design significantly reduces the time complexity of locating elements compared to the original method. By leveraging this structure, the system avoids the overhead of sequentially traversing all datablocks, making the retrieval process highly efficient and scalable, especially in environments with a large number of entries.

```c
void hash_insert(const char *, uint , uint );
void hash_access(const char *, uint *, uint *);
void hash_invalidate(const char *);
void hash_init();
```

1.2 skipelem()

To evaluate the functionality of skipelem(), I created test cases, and the function performed well in all scenarios. Based on these results, I concluded that it is appropriate to utilize this function as part of the implementation. To adapt this functionality for full-path indexing, the namex function was modified accordingly.

1.3 On memeory inode freelist

Currently, when allocating an on-memory inode table or searching for it through iget(), a sequential search is used to return the corresponding itable entry or allocate a new one. I optimized this by implementing a table-based approach for direct access, eliminating the need for sequential search. Additionally, I introduced a freelist to quickly locate and allocate free entries, improving efficiency.

2 Algorithm design

2.1 sleeplock()

Currently, the sleep lock iterates through all processes to wake up processes sleeping on the specified channel, marking them as RUNNABLE. This sequential search through all processes is inefficient. I optimized this by adding a counter to keep track of the number of sleeping processes. If no processes are sleeping, the iteration is skipped. Additionally, if the counter reaches the specified number, the loop breaks early, avoiding a full iteration through all processes.

2.2 namex()

The namex function is currently used extensively across the system. It parses the user-provided path to return the inode of the target file or directory. Additionally, when creating a new entry, it returns the inode of the parent directory while storing the entry's name in the name variable.

For full-path indexing, the first functionality remains unchanged; the function should still return the inode by comparing the full path against the entries' names. However, when creating a new entry, while the parent directory's inode is still returned, the name variable should contain the full path. Based on this requirement, I planned and modified the namex function accordingly.

When an absolute path is provided as input, it is used as is. For a relative path, the current working directory (cwd) is utilized to combine with the relative input, converting it into an absolute path using skipelem(). This absolute path is then passed to dirlookup() to retrieve and return the corresponding inode.

2.3 safestrcpy()

The safestrcpy function copies up to n bytes from t to s. Originally, it returned os, which is the starting address of s. However, in the namex() function, I needed to process entries like `".."`, `"."`, and other directory entries correctly using skipelem() and append them to the absolute path one element at a time. To facilitate this, I modified the function by updating os = s to return the position of the final "\0" character, making it more suitable for this use case.

```c
char*
safestrcpy(char *s, const char *t, int n)
{
  char *os;

  os = s;
  if(n <= 0)
    return os;
  while(--n > 0 && (*s++ = *t++) != 0)
    ;
  *s = 0;
  os = s;
  return os;
}
```

2.4 chdir()

Each process has its own current working directory (cwd), which corresponds to its own pwd. The namei function interprets the input path as an absolute path and stores it in pwd. However, the current implementation of namei only returns the inode and does not provide the name. This can lead to situations where the directory's datablocks need to be re-scanned from the beginning to retrieve the name.

To address this issue, I created a new function, namein, which not only returns the inode but also stores the full path name in the name variable. This ensures efficient handling of both the inode and the full path without requiring redundant operations.

```c
struct inode*
namein(char *path, char *name) //
{
  return namex(path, 0, name);
}
```

2.5 Conercases

1. To perform unlink, it is necessary to check whether a directory is empty using the isdirempty() function. Currently, all subdirectory datablocks are empty. Therefore, child nodes must be located within the root datablock by searching for their names in the directory. To address this, I modified the isdirempty() function to incorporate this functionality.

2. The name of each entry has a maximum length of 106 bytes. When combining a relative path with an absolute path, the resulting path could exceed 106 bytes. To accommodate this, the buffer size used in namex was increased to be larger than 106 bytes.

3. Some name-related buffer sizes were left as DIRSIZ, which caused errors during the iref test. I also wrote code to track the reference count separately, but no issues were found there. It turned out the problem was caused by the path size becoming too large. To resolve this, I replaced all instances of DIRSIZ with MAXPATH.

2.6 Efficiency

The hashtable significantly reduced the time required for dirlookup(). The original dirlookup() iterated through the directory's datablocks to check if the specified name existed. As the root directory's datablocks have grown larger, the overhead from this iteration has become more pronounced. For example, locating an element in a directory 5 levels deep required sequential traversal starting from the README entry. To address this issue, a hashtable was implemented.

As a result, the time per file lookup, which previously took 24,000 cycles per file according to fsperf, has now been reduced to just 993 cycles per file, demonstrating a significant improvement in performance.

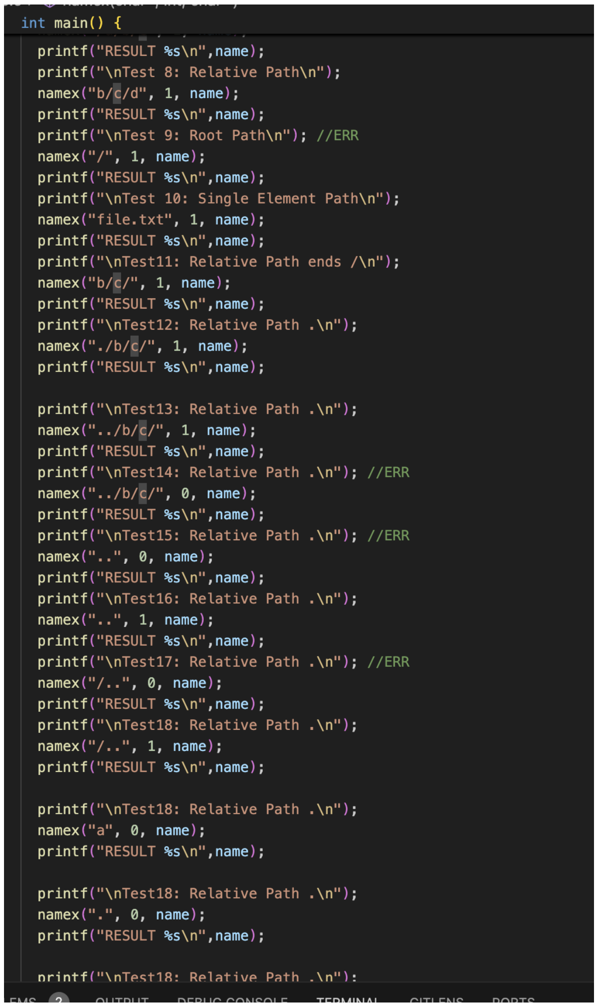
3 Testing and validation

1. I created test cases where the path name length was 106, and the relative path length of the command was also 106. These cases were used to test the system under these specific conditions.

2. I created a new C file to test and verify that the namex function I implemented works correctly for various paths.

3. When following the cat example in the README as is, I found that giving a relative path (e.g., cd jinsoo) as input to a command within a subdirectory did not work correctly. I resolved the issue by adding additional conditional statements to the namex function.

4. In paths, `"/"` is typically used as a delimiter between directories. However, at the root, `"/"` itself represents the directory, requiring a different handling approach. Initially, I did not account for this, which caused examples like `"/."` to be processed incorrectly. I fixed this issue by modifying the namex function accordingly.



```c
int main() {
    printf("RESULT %s\n",name);
    printf("\nTest 8: Relative Path\n");
    namex("b/c/d", 1, name);
    printf("RESULT %s\n",name);
    printf("\nTest 9: Root Path\n"); //ERR
    namex("/", 1, name);
    printf("RESULT %s\n",name);
    printf("\nTest 10: Single Element Path\n");
    namex("file.txt", 1, name);
    printf("RESULT %s\n",name);
    printf("\nTest11: Relative Path ends /\n");
    namex("b/c/", 1, name);
    printf("RESULT %s\n",name);
    printf("\nTest12: Relative Path .\n");
    namex("./b/c/", 1, name);
    printf("RESULT %s\n",name);

    printf("\nTest13: Relative Path .\n");
    namex("../b/c/", 1, name);
    printf("RESULT %s\n",name);
    printf("\nTest14: Relative Path .\n"); //ERR
    namex("../b/c/", 0, name);
    printf("RESULT %s\n",name);
    printf("\nTest15: Relative Path .\n"); //ERR
    namex("..", 0, name);
    printf("RESULT %s\n",name);
    printf("\nTest16: Relative Path .\n");
    namex("..", 1, name);
    printf("RESULT %s\n",name);
    printf("\nTest17: Relative Path .\n"); //ERR
    namex("/..", 0, name);
    printf("RESULT %s\n",name);
    printf("\nTest18: Relative Path .\n");
    namex("/..", 1, name);
    printf("RESULT %s\n",name);

    printf("\nTest18: Relative Path .\n");
    namex("a", 0, name);
    printf("RESULT %s\n",name);

    printf("\nTest18: Relative Path .\n");
    namex(".", 0, name);
    printf("RESULT %s\n",name);

    printf("\nTest18: Relative Path .\n");
```

Figure 1: Test Cases