

inode number

```
1  #include <stdio.h>
2  #include <sys/stat.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 2) {
6          printf("Usage: %s <filename>\n", argv[0]);
7          return 1;
8      }
9
10     struct stat fileStat;
11
12     if (stat(argv[1], &fileStat) == -1) {
13         perror("stat");
14         return 1;
15     }
16
17     printf("Inode number: %lu\n", fileStat.st_ino);
18
19     return 0;
20 }
```

```
find / -inum <inode_number> -exec cat {} \;
```

Virtual File System (VFS)

```
1  struct {
2      struct spinlock lock;
3      struct file file[NR_OPEN_FILES];
4  } ftable;
5
6  struct file {
7      struct inode *inode;
8      int ref;
9      int flags;
10     off_t offset;
11 };
```

```
1  struct inode {
2      uint dev;
3      uint inum;
4      int ref;
5      struct sleeplock lock;
6      int valid;
7
8      short type;
9      short major;
10     short minor;
11     short nlink;
12     uint size;
13     uint addrs[NDIRECT+1];
14 };
```

strace

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/wait.h>
5
6  #define NR_GLOBAL_FILES 1024
7  #define NR_OPEN_FILES 256
8
9  struct spinlock {
10     int locked;
11 };
12
13 struct inode {};
14
15 struct file {
16     struct inode *inode;
17     int ref;
18     int flags;
19     off_t offset;
20 };
21
22 struct ftable {
23     struct spinlock lock;
24     struct file file[NR_GLOBAL_FILES];
25 } global_ftable;
26
```

```

27 struct fdtable {
28     struct file *files[NR_OPEN_FILES];
29 };
30
31 void spinlock_acquire(struct spinlock *lock) {
32     while (__sync_lock_test_and_set(&lock->locked, 1)) {}
33 }
34
35 void spinlock_release(struct spinlock *lock) {
36     __sync_lock_release(&lock->locked);
37 }
38
39 void ftable_init() {
40     global_ftable.lock.locked = 0;
41     for (int i = 0; i < NR_GLOBAL_FILES; i++) {
42         global_ftable.file[i].inode = NULL;
43         global_ftable.file[i].ref = 0;
44     }
45 }
46
47 void fdtable_init(struct fdtable *fdtable) {
48     for (int i = 0; i < NR_OPEN_FILES; i++) {
49         fdtable->files[i] = NULL;
50     }
51 }
52
53 struct file *ftable_alloc(struct inode *inode, int flags) {
54     spinlock_acquire(&global_ftable.lock);
55     for (int i = 0; i < NR_GLOBAL_FILES; i++) {
56         if (global_ftable.file[i].ref == 0) {
57             global_ftable.file[i].inode = inode;
58             global_ftable.file[i].flags = flags;
59             global_ftable.file[i].offset = 0;
60             global_ftable.file[i].ref = 1;
61             spinlock_release(&global_ftable.lock);
62             return &global_ftable.file[i];
63         }
64     }
65     spinlock_release(&global_ftable.lock);
66     return NULL;
67 }
68
69 int fdtable_alloc(struct fdtable *fdtable, struct file *file) {
70     spinlock_acquire(&global_ftable.lock);
71     for (int fd = 0; fd < NR_OPEN_FILES; fd++) {
72         if (fdtable->files[fd] == NULL) {

```

```

73     fdtable->files[fd] = file;
74     file->ref++;
75     spinlock_release(&global_fhtable.lock);
76     return fd;
77 }
78 }
79 spinlock_release(&global_fhtable.lock);
80 return -1;
81 }
82
83 void fdtable_release(struct fdtable *fdtable, int fd) {
84     if (fd < 0 || fd >= NR_OPEN_FILES || fdtable->files[fd] == NULL) return;
85     spinlock_acquire(&global_fhtable.lock);
86     struct file *file = fdtable->files[fd];
87     file->ref--;
88     if (file->ref == 0) {
89         file->inode = NULL;
90         file->flags = 0;
91         file->offset = 0;
92     }
93     fdtable->files[fd] = NULL;
94     spinlock_release(&global_fhtable.lock);
95 }
96
97 void open_file(struct fdtable *fdtable) {
98     struct inode dummy_inode;
99     struct file *new_file = ftable_alloc(&dummy_inode, 0);
100     if (new_file) {
101         int fd = fdtable_alloc(fdtable, new_file);
102         printf("File descriptor: %d\n", fd);
103     }
104 }
105
106 int main() {
107     struct fdtable process_fdtable1, process_fdtable2;
108     ftable_init();
109     fdtable_init(&process_fdtable1);
110     fdtable_init(&process_fdtable2);
111
112     pid_t pid = fork();
113
114     if (pid == 0) {
115         open_file(&process_fdtable1);
116         _exit(0);
117     } else if (pid > 0) {
118         open_file(&process_fdtable2);

```

```
119     wait(NULL);
120 }
121
122     return 0;
123 }
124
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main() {
7      char buf1[100];
8      char buf2[100];
9
10     int fd = open("file", O_RDONLY);
11     if (fd == -1) {
12         perror("open");
13         exit(1);
14     }
15     read(fd, buf1, 100);
16     read(fd, buf1, 100);
17     read(fd, buf1, 100);
18     read(fd, buf1, 100);
19     close(fd);
20
21     int fd1 = open("file", O_RDONLY);
22     int fd2 = open("file", O_RDONLY);
23     if (fd1 == -1 || fd2 == -1) {
24         perror("open");
25         exit(1);
26     }
27     read(fd1, buf1, 100);
28     read(fd2, buf2, 100);
29     close(fd1);
30     close(fd2);
31
32     fd = open("file", O_RDONLY);
33     if (fd == -1) {
34         perror("open");
35         exit(1);
36     }
```

```
37     lseek(fd, 200, SEEK_SET);
38     read(fd, buf1, 50);
39     close(fd);
40
41     return 0;
42 }
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5
6  int main() {
7      char buf[100];
8      int fd = open("file", O_RDONLY);
9      if (fd == -1) {
10         perror("open");
11         exit(1);
12     }
13     pread(fd, buf, 100, 0);
14     pread(fd, buf, 100, 100);
15     pread(fd, buf, 100, 200);
16     close(fd);
17     return 0;
18 }
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <assert.h>
4  #include <stdio.h>
5  #include <sys/wait.h>
6
7  int main(int argc, char *argv[]) {
8      int fd = open("file.txt", O_RDONLY);
9      assert(fd >= 0);
10     int pid = fork();
11     if (pid == 0) {
12         int off = lseek(fd, 100, SEEK_SET);
13         printf("Child process: current offset is %d\n", off);
14     } else if (pid > 0) {
```

```
15     (void) wait(NULL);
16     int curr_off = (int) lseek(fd, 0, SEEK_CUR);
17     printf("Parent process: current offset is %d\n", curr_off);
18 }
19 return 0;
20 }
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  int main() {
6      int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
7      if (fd == -1) exit(1);
8      dup2(fd, 1);
9      close(fd);
10     execlp("ls", "ls", NULL);
11     return 0;
12 }
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3
4  int main() {
5      int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
6      int fd_dup = dup(fd);
7      write(fd, "Hello, ", 7);
8      write(fd_dup, "World!", 6);
9      close(fd);
10     close(fd_dup);
11     return 0;
12 }
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4
5  int main() {
```

```
6   int fd1 = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
7   int fd2 = open("output.txt", O_WRONLY);
8
9   write(fd1, "Hello, ", 7);
10  lseek(fd2, 0, SEEK_SET);
11  write(fd2, "World!", 6);
12
13  close(fd1);
14  close(fd2);
15
16  return 0;
17 }
```

```
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <sys/mman.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <stdio.h>
7  #include <assert.h>
8  //msync might be required
9  typedef struct Node {
10     int key;
11     int left;
12     int right;
13 } Node;
14
15 typedef struct {
16     int root;
17     int next_free;
18     Node nodes[];
19 } PersistentBST;
20
21 void insert(PersistentBST *tree, int key, int max_nodes) {
22     if (tree->next_free >= max_nodes) return;
23     int curr = tree->root;
24     int *link = &tree->root;
25     while (curr != -1) {
26         if (key < tree->nodes[curr].key) {
27             link = &tree->nodes[curr].left;
28             curr = tree->nodes[curr].left;
29         } else if (key > tree->nodes[curr].key) {
30             link = &tree->nodes[curr].right;
```



```

31     curr = tree->nodes[curr].right;
32 } else {
33     return;
34 }
35 }
36 int idx = tree->next_free++;
37 *link = idx;
38 tree->nodes[idx].key = key;
39 tree->nodes[idx].left = -1;
40 tree->nodes[idx].right = -1;
41 }
42
43 void inorder(PersistentBST *tree, int curr) {
44     if (curr == -1) return;
45     inorder(tree, tree->nodes[curr].left);
46     printf("%d ", tree->nodes[curr].key);
47     inorder(tree, tree->nodes[curr].right);
48 }
49
50 int main(int argc, char *argv[]) {
51     int fd = open("bst.bin", O_RDWR | O_CREAT, 0644);
52     int file_size = sizeof(PersistentBST) + 100 * sizeof(Node);
53     ftruncate(fd, file_size);
54     PersistentBST *tree = mmap(NULL, file_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
55     assert(tree != MAP_FAILED);
56     if (tree->next_free == 0) {
57         tree->root = -1;
58         tree->next_free = 0;
59     }
60     for (int i = 1; i < argc; i++) {
61         if (strcmp(argv[i], "print") == 0) {
62             inorder(tree, tree->root);
63             printf("\n");
64         } else {
65             int key = atoi(argv[i]);
66             insert(tree, key, 100);
67         }
68     }
69     munmap(tree, file_size);
70     close(fd);
71     return 0;
72 }

```

