inode number

```c
#include <stdio.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: %s <filename>\n", argv[0]);
    return 1;
  }

  struct stat fileStat;

  if (stat(argv[1], &fileStat) == -1) {
    perror("stat");
    return 1;
  }

  printf("Inode number: %lu\n", fileStat.st_ino);

  return 0;
}
```

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

Virtual File System (VFS)

```c
struct {
  struct spinlock lock;
  struct file file[NR_OPEN_FILES];
} ftable;

struct file {
  struct inode *inode;
  int ref;
  int flags;
  off_t offset;
};
```

```c
struct inode {
  uint dev;
  uint inum;
  int ref;
  struct sleeplock lock;
  int valid;

  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};
```

`strace`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

#define NR_GLOBAL_FILES 1024
#define NR_OPEN_FILES 256

struct spinlock {
  int locked;
};

struct inode {};

struct file {
  struct inode *inode;
  int ref;
  int flags;
  off_t offset;
};

struct ftable {
  struct spinlock lock;
  struct file file[NR_GLOBAL_FILES];
} global_ftable;

```

```c
struct fdtable {
  struct file *files[NR_OPEN_FILES];
};

void spinlock_acquire(struct spinlock *lock) {
  while (__sync_lock_test_and_set(&lock->locked, 1)) {}
}

void spinlock_release(struct spinlock *lock) {
  __sync_lock_release(&lock->locked);
}

void ftable_init() {
  global_ftable.lock.locked = 0;
  for (int i = 0; i < NR_GLOBAL_FILES; i++) {
    global_ftable.file[i].inode = NULL;
    global_ftable.file[i].ref = 0;
  }
}

void fdtable_init(struct fdtable *fdtable) {
  for (int i = 0; i < NR_OPEN_FILES; i++) {
    fdtable->files[i] = NULL;
  }
}

struct file *ftable_alloc(struct inode *inode, int flags) {
  spinlock_acquire(&global_ftable.lock);
  for (int i = 0; i < NR_GLOBAL_FILES; i++) {
    if (global_ftable.file[i].ref == 0) {
      global_ftable.file[i].inode = inode;
      global_ftable.file[i].flags = flags;
      global_ftable.file[i].offset = 0;
      global_ftable.file[i].ref = 1;
      spinlock_release(&global_ftable.lock);
      return &global_ftable.file[i];
    }
  }
  spinlock_release(&global_ftable.lock);
  return NULL;
}

int fdtable_alloc(struct fdtable *fdtable, struct file *file) {
  spinlock_acquire(&global_ftable.lock);
  for (int fd = 0; fd < NR_OPEN_FILES; fd++) {
    if (fdtable->files[fd] == NULL) {
      fdtable->files[fd] = file;
      file->ref++;
      spinlock_release(&global_ftable.lock);
      return fd;
    }
  }
  spinlock_release(&global_ftable.lock);
  return -1;
}

void fdtable_release(struct fdtable *fdtable, int fd) {
  if (fd < 0 || fd >= NR_OPEN_FILES || fdtable->files[fd] == NULL) return;
  spinlock_acquire(&global_ftable.lock);
  struct file *file = fdtable->files[fd];
  file->ref--;
  if (file->ref == 0) {
    file->inode = NULL;
    file->flags = 0;
    file->offset = 0;
  }
  fdtable->files[fd] = NULL;
  spinlock_release(&global_ftable.lock);
}

void open_file(struct fdtable *fdtable) {
  struct inode dummy_inode;
  struct file *new_file = ftable_alloc(&dummy_inode, 0);
  if (new_file) {
    int fd = fdtable_alloc(fdtable, new_file);
    printf("File descriptor: %d\n", fd);
  }
}

int main() {
  struct fdtable process_fdtable1, process_fdtable2;
  ftable_init();
  fdtable_init(&process_fdtable1);
  fdtable_init(&process_fdtable2);

  pid_t pid = fork();

  if (pid == 0) {
    open_file(&process_fdtable1);
    _exit(0);
  } else if (pid > 0) {
    open_file(&process_fdtable2);
    wait(NULL);
  }

  return 0;
}

```

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  char buf1[100];
  char buf2[100];

  int fd = open("file", O_RDONLY);
  if (fd == -1) {
    perror("open");
    exit(1);
  }
  read(fd, buf1, 100);
  read(fd, buf1, 100);
  read(fd, buf1, 100);
  read(fd, buf1, 100);
  close(fd);

  int fd1 = open("file", O_RDONLY);
  int fd2 = open("file", O_RDONLY);
  if (fd1 == -1 || fd2 == -1) {
    perror("open");
    exit(1);
  }
  read(fd1, buf1, 100);
  read(fd2, buf2, 100);
  close(fd1);
  close(fd2);

  fd = open("file", O_RDONLY);
  if (fd == -1) {
    perror("open");
    exit(1);
  }
  lseek(fd, 200, SEEK_SET);
  read(fd, buf1, 50);
  close(fd);

  return 0;
}
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
  char buf[100];
  int fd = open("file", O_RDONLY);
  if (fd == -1) {
    perror("open");
    exit(1);
  }
  pread(fd, buf, 100, 0);
  pread(fd, buf, 100, 100);
  pread(fd, buf, 100, 200);
  close(fd);
  return 0;
}
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <assert.h>
#include <stdio.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int fd = open("file.txt", O_RDONLY);
  assert(fd >= 0);
  int pid = fork();
  if (pid == 0) {
    int off = lseek(fd, 100, SEEK_SET);
    printf("Child process: current offset is %d\n", off);
  } else if (pid > 0) {
        (void) wait(NULL);
        int curr_off = (int) lseek(fd, 0, SEEK_CUR);
        printf("Parent process: current offset is %d\n", curr_off);
    }
    return 0;
}
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
  int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd == -1) exit(1);
  dup2(fd, 1);
  close(fd);
  execlp("ls", "ls", NULL);
  return 0;
}
```

```c
#include <fcntl.h>
#include <unistd.h>

int main() {
  int fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  int fd_dup = dup(fd);
  write(fd, "Hello, ", 7);
  write(fd_dup, "World!", 6);
  close(fd);
  close(fd_dup);
  return 0;
}
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
  int fd1 = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  int fd2 = open("output.txt", O_WRONLY);

  write(fd1, "Hello, ", 7);
  lseek(fd2, 0, SEEK_SET);
  write(fd2, "World!", 6);

  close(fd1);
  close(fd2);

  return 0;
}
```

```c
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <assert.h>
//msync might be required
typedef struct Node {
  int key;
  int left;
  int right;
} Node;

typedef struct {
  int root;
  int next_free;
  Node nodes[];
} PersistentBST;

void insert(PersistentBST *tree, int key, int max_nodes) {
  if (tree->next_free >= max_nodes) return;
  int curr = tree->root;
  int *link = &tree->root;
  while (curr != -1) {
    if (key < tree->nodes[curr].key) {
      link = &tree->nodes[curr].left;
      curr = tree->nodes[curr].left;
    } else if (key > tree->nodes[curr].key) {
      link = &tree->nodes[curr].right;
      curr = tree->nodes[curr].right;
    } else {
      return;
    }
  }
  int idx = tree->next_free++;
  *link = idx;
  tree->nodes[idx].key = key;
  tree->nodes[idx].left = -1;
  tree->nodes[idx].right = -1;
}

void inorder(PersistentBST *tree, int curr) {
  if (curr == -1) return;
  inorder(tree, tree->nodes[curr].left);
  printf("%d ", tree->nodes[curr].key);
  inorder(tree, tree->nodes[curr].right);
}

int main(int argc, char *argv[]) {
  int fd = open("bst.bin", O_RDWR | O_CREAT, 0644);
  int file_size = sizeof(PersistentBST) + 100 * sizeof(Node);
  ftruncate(fd, file_size);
  PersistentBST *tree = mmap(NULL, file_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
  assert(tree != MAP_FAILED);
  if (tree->next_free == 0) {
    tree->root = -1;
    tree->next_free = 0;
  }
  for (int i = 1; i < argc; i++) {
    if (strcmp(argv[i], "print") == 0) {
      inorder(tree, tree->root);
      printf("\n");
    } else {
      int key = atoi(argv[i]);
      insert(tree, key, 100);
    }
  }
  munmap(tree, file_size);
  close(fd);
  return 0;
}
```