



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт по лабораторной работе №10 по курсу "Операционные системы"

Тема Буферизованный и не буферизованный ввод-вывод.

Студент Нарандаев Д.С.

Группа ИУ7-62Б

Преподаватель Рязанова Н.Ю.

Дата:
15.06.2023

Подпись:

Москва — 2023 г.

1. Программы

1.1. Первая программа

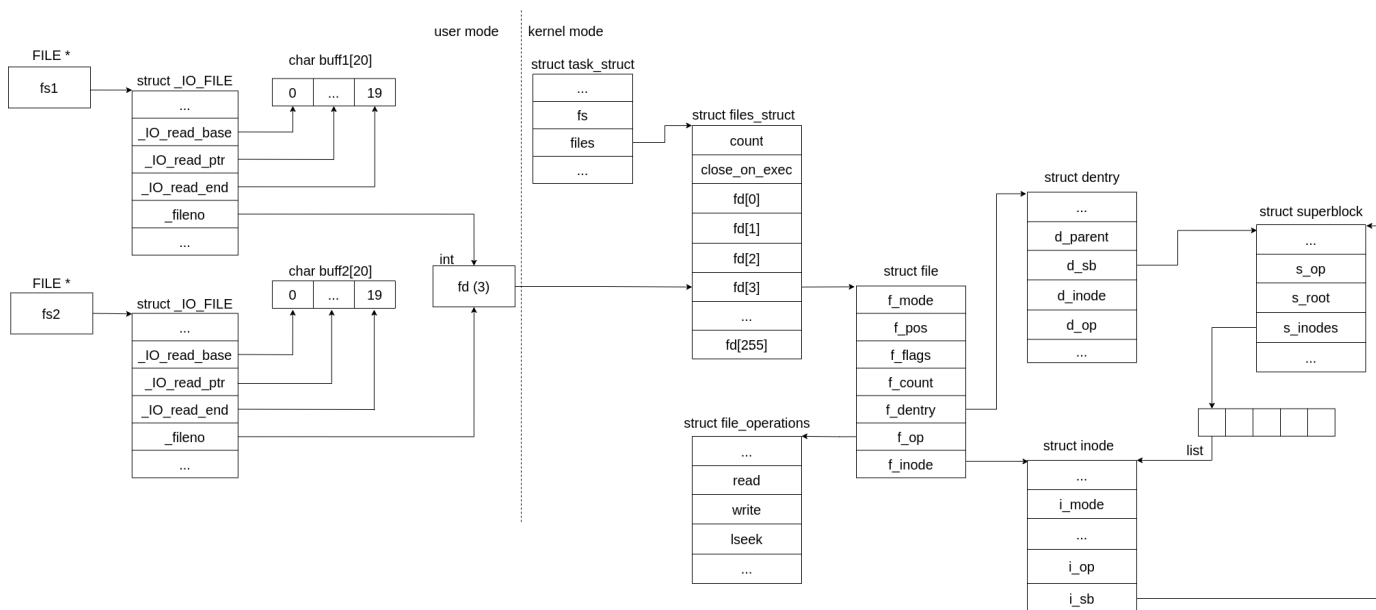


Рисунок 1.1 — Связь структур

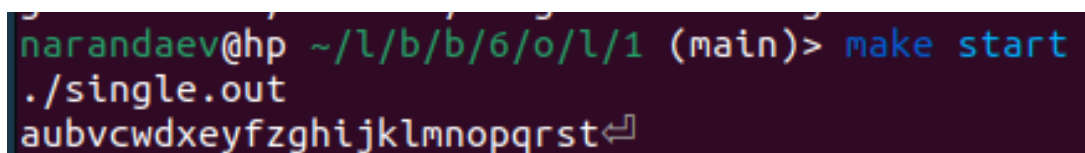
Однопоточная реализация

```
1 #include <stdio.h>
2 #include <fcntl.h>
3
4 int main() {
5     int fd = open("alphabet.txt", O_RDONLY);
6
7     FILE *fs1 = fdopen(fd, "r");
8     char buff1[20];
9     setvbuf(fs1, buff1, _IOFBF, 20);
10 }
```

```

11 FILE *fs2 = fdopen(fd, "r");
12 char buff2[20];
13 setvbuf(fs2, buff2, _IOFBF, 20);
14
15 int flag1 = 1, flag2 = 2;
16 while (flag1 == 1 || flag2 == 1) {
17     char c;
18     flag1 = fscanf(fs1, "%c", &c);
19     if (flag1 == 1)
20         fprintf(stdout, "%c", c);
21     flag2 = fscanf(fs2, "%c", &c);
22
23     if (flag2 == 1)
24         fprintf(stdout, "%c", c);
25 }
26 return 0;
27 }

```



```

narandaev@hp ~/l/b/b/6/o/l/1 (main)> make start
./single.out
aubvcwdxeyfzghijklmnopqrst

```

Рисунок 1.2 — Вывод программы

С помощью системного вызова `open()` создается дескриптор открытого файла. Системный вызов `open()` возвращает индекс в массиве `fd_array` структуры `files_struct`. Библиотечная функция `fdopen()` возвращает указатели на `struct FILE` (`fs1` и `fs2`), которые ссылаются на дескриптор, созданный системным вызовом `open()`. Далее создаются буферы `buff1` и `buff2` размером 20 байт. Для дескрипторов `fs1` и `fs2` функцией `setvbuf()` задаются соответствующие буферы и тип буферизации `_IOFBF`.

Далее `fscanf()` выполняется в цикле поочередно для `fs1` и `fs2`. При вызове `fscanf()` для `fs1` в буфер `buff1` считываются первые 20 символов. Значение `f_pos` в структуре `struct file` открытого увеличится на 20. В переменную `c` записывается символ 'a' и выводится с помощью `fprintf()`. При вызове `fscanf()` для `fs2` в буфер `buff2` считываются оставшиеся 6 символов.

В цикле символы из `buff1` и `buff2` будут поочередно выводиться до тех пор, пока символы в одном из буферов не закончатся. Тогда на экран будут последовательно выведены оставшиеся символы из другого буфера.

Многопоточная реализация

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 #include <pthread.h>
4
5 void *task(void *fd) {
6     int flag = 1;
7     char c;
8
9     FILE *fs = fdopen((int)fd, "r");
10    char buff[20];
11    setvbuf(fs, buff, _IOFBF, 20);
12
13    while (flag == 1) {
14        flag = fscanf(fs, "%c", &c);
15        if (flag == 1)
16            fprintf(stdout, "%c", c);
17    }
18 }
19
20 int main(){
21     int fd = open("alphabet.txt", O_RDONLY);
22
23     pthread_t thids[2];
24     pthread_create(thids, NULL, task, (void *)fd);
25     pthread_create(thids+1, NULL, task, (void *)fd);
26
27     pthread_join(thids[0], NULL);
28     pthread_join(thids[1], NULL);
29     return 0;
30 }
```

```

narandaev@hp ~/l/b/b/6/o/l/1 (main)> make start-
./multithread.out
abcdefghijklmnopqrstuvwxyz

```

Рисунок 1.3 — Вывод программы

В многопоточной реализации в цикле символы из `buff1` и `buff2` будут выводиться параллельно. Обе структуры `_IO_FILE` имеют одинаковый файловый дескриптор, поэтому в файл будет записан весь алфавит, но порядок символов не может быть гарантировано определён.

1.2. Вторая программа

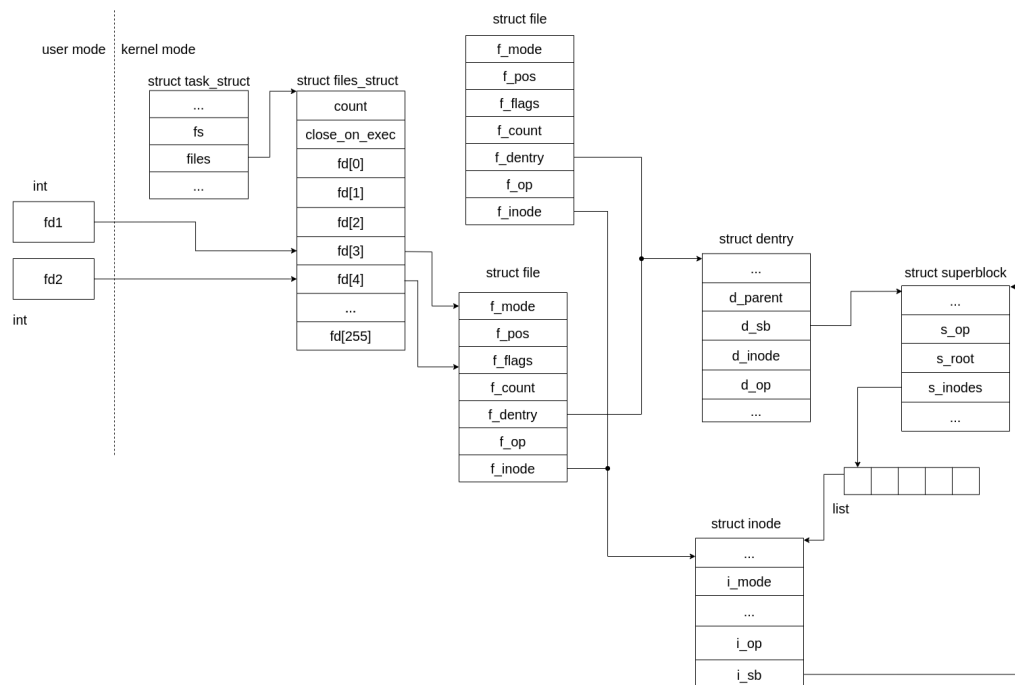


Рисунок 1.4 — Связь структур

Однопоточная реализация

```

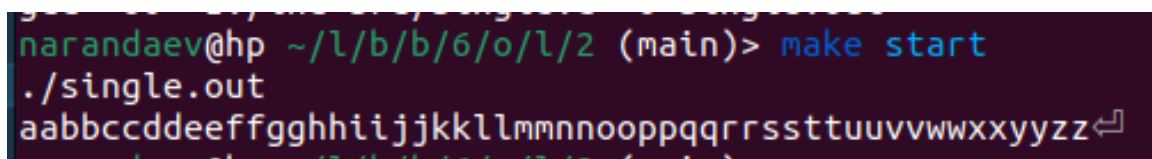
1 #include <fcntl.h>
2 #include <unistd.h>
3
4 int main() {
5     char c;

```

```

6   int fd1 = open("alphabet.txt", O_RDONLY);
7   int fd2 = open("alphabet.txt", O_RDONLY);
8   int flag1 = 1, flag2 = 1;
9   while(flag1 && flag2) {
10      if ((flag1 = read(fd1, &c, 1)))
11         write(1, &c, 1);
12      if ((flag2 = read(fd2, &c, 1)))
13         write(1, &c, 1);
14  }
15  return 0;
16 }

```



```

narandaev@hp ~/l/b/b/6/o/l/2 (main)> make start
./single.out
aabbccddeeffgghhiijjkkllmmnnnooppqrrssttuuvvwwxxyyzz

```

Рисунок 1.5 — Вывод программы

Один и тот же файл открывается дважды, следовательно создается две структуры `struct file`, в которых `f_inode` указывает на один и тот же `struct inode`. У каждого файлового дескриптора своя позиция (`f_pos`), поэтому при чтении из `fd1` и `fd2` файл прочитывается от начала до конца. Используя каждый дескриптор, читаем по одному символу и выводим его, в результате получаем, что каждый символ выводится дважды.

Многопоточная реализация

```

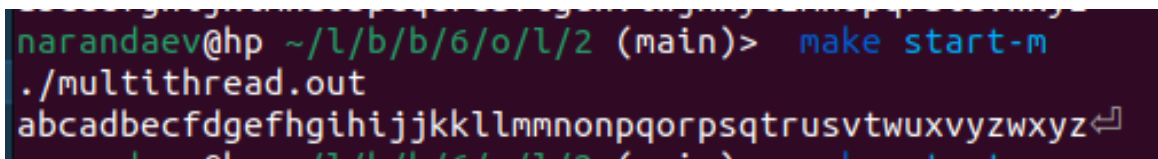
1  #include <fcntl.h>
2  #include <unistd.h>
3  #include <pthread.h>
4
5  void *task() {
6      int fd = open("alphabet.txt", O_RDONLY);
7      int flag = 1;
8      char c;
9
10     while (flag) {
11         if ((flag=read(fd, &c, 1)))

```

```

12     write(1,&c,1);
13 }
14 }
15
16 int main(){
17     pthread_t thids[2];
18     pthread_create(thids,NULL,task,NULL);
19     pthread_create(thids+1,NULL,task,NULL);
20
21     pthread_join(thids[0],NULL);
22     pthread_join(thids[1],NULL);
23     return 0;
24 }

```



```

narandaev@hp ~/l/b/b/6/o/l/2 (main)> make start-m
./multithread.out
abcadbefcdgefghijhjjkllmmnonpqorpsqtrusvtwuxvyzwxxyz

```

Рисунок 1.6 — Вывод программы

Каждый поток читает и пишет в stdout. Так как оба потока пишут одновременно, алфавит перемешивается. Из-за того, что запись происходит асинхронно, гарантировано предсказать, как именно будет выглядеть вывод, невозможно.

1.3. Третья программа, версия с использованием stdio.h

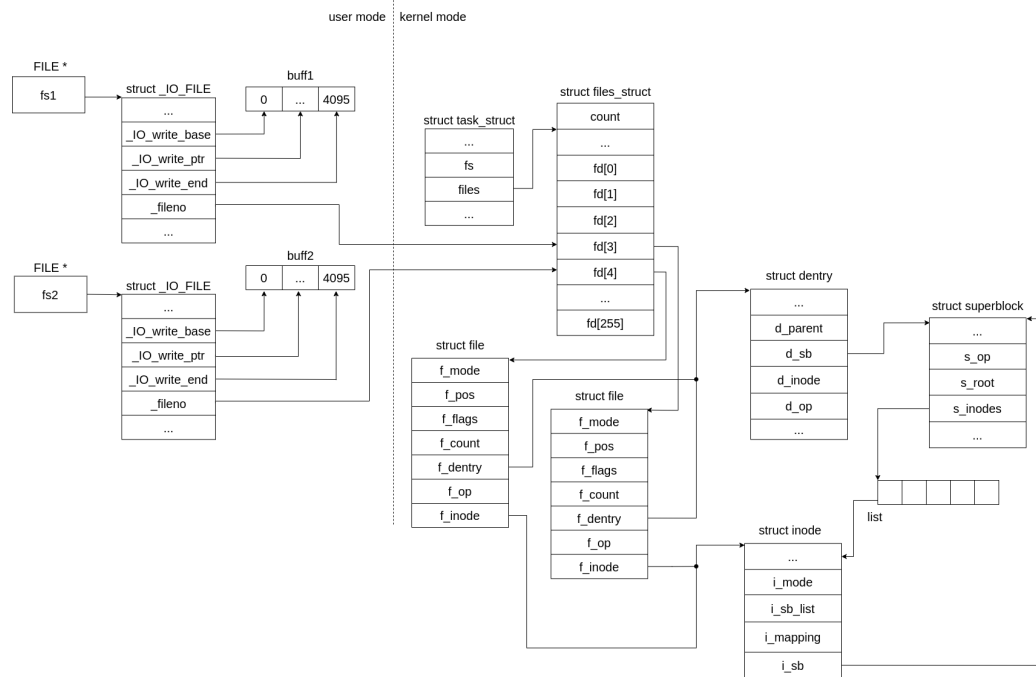


Рисунок 1.7 — Связь структур

Однопоточная реализация

```

1  #include <stdio.h>
2  #include <sys/stat.h>
3  #include <fcntl.h>
4
5  #define PRINT_STAT(path, action) \
6    do { \
7        stat(path, &statbuf); \
8        fprintf(stdout, action ": inode num = %ld, size = %ld, \
9            blksize = %ld\n", \
10            statbuf.st_ino, statbuf.st_size, \
11            statbuf.st_blksize); \
12    } while (0)
13
14 int main() {
15     struct stat statbuf;
16     FILE *file1 = fopen("output.txt", "w");

```



```

16 PRINT_STAT("output.txt", "fopen file1");
17 FILE *file2 = fopen("output.txt", "w");
18 PRINT_STAT("output.txt", "fopen file2");
19
20 for (char ch='a'; ch<='z'; ch++) {
21     if (ch % 2)
22         fprintf(file1,"%c",ch);
23     else
24         fprintf(file2,"%c",ch);
25     PRINT_STAT("output.txt", "fprintf");
26 }
27
28 fclose(file1);
29 PRINT_STAT("output.txt", "fclose file1");
30 fclose(file2);
31 PRINT_STAT("output.txt", "fclose file2");
32 return 0;
33 }

```

```

narandaev@hp ~/l/b/b/6/o/l/3 (main)> make start
./single.out
fopen file1: inode num = 6687572, size = 0, blksize = 4096
fopen file2: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fprintf: inode num = 6687572, size = 0, blksize = 4096
fclose file1: inode num = 6687572, size = 13, blksize = 4096
fclose file2: inode num = 6687572, size = 13, blksize = 4096
narandaev@hp ~/l/b/b/6/o/l/3 (main)> cat ./output.txt
bdfhjlnprtvxz

```

Рисунок 1.8 — Вывод программы

Файл открывается дважды с помощью `fopen`. Содержимое записывается из буфера в файл в следующих случаях:

- буфер уже заполнен и происходит попытка очередной записи в буфер;
- вызов `fflush()`;
- вызов функций `close()`, `fclose()`.

В данном случае запись в файл происходит в результате вызова функции `fclose()`. При вызове `fclose()` для `fs1` буфер для `fs1` записывается в файл. При вызове `fclose()` для `fs2`, все содержимое файла очищается, а в файл записывается содержимое буфера для `fs2`. В итоге произошла потеря данных, в файле окажется только содержимое буфера для `fs2`.

Многопоточная реализация

```

1 #include <stdio.h>
2 #include <fcntl.h>

```

```

3 #include <pthread.h>
4
5 void *task(void *payload) {
6     int n = (int)payload;
7     FILE *file = fopen("output.txt", "w");
8     for (char ch = 'a'; ch <= 'z'; ch++) {
9         if (ch % 2 == n)
10             fprintf(file, "thread_ind = %d: char = %c\n", n, ch);
11     }
12     fclose(file);
13     return 0;
14 }
15
16 int main() {
17     pthread_t thids[2];
18     pthread_create(&thids[0], NULL, task, (void *)0);
19     pthread_create(&thids[1], NULL, task, (void *)1);
20     pthread_join(thids[0], NULL);
21     pthread_join(thids[1], NULL);
22     return 0;
23 }

```

```
narandaev@hp ~/l/b/b/6/o/l/3 (main)> make start-m
./multithread.out
narandaev@hp ~/l/b/b/6/o/l/3 (main)> cat ./output.txt
thread_ind = 1: char = a
thread_ind = 1: char = c
thread_ind = 1: char = e
thread_ind = 1: char = g
thread_ind = 1: char = i
thread_ind = 1: char = k
thread_ind = 1: char = m
thread_ind = 1: char = o
thread_ind = 1: char = q
thread_ind = 1: char = s
thread_ind = 1: char = u
thread_ind = 1: char = w
thread_ind = 1: char = y
```

Рисунок 1.9 — Вывод программы

В многопоточной программе работа с файлом производится аналогично однопоточной программе. Если вызывать `fclose()` в дополнительном потоке, то порядок вывода символов не может быть гарантировано определён, так как нельзя предсказать заранее, какой поток последним вызовет `fclose()`.

1.4. Третья программа, версия с системными вызовами

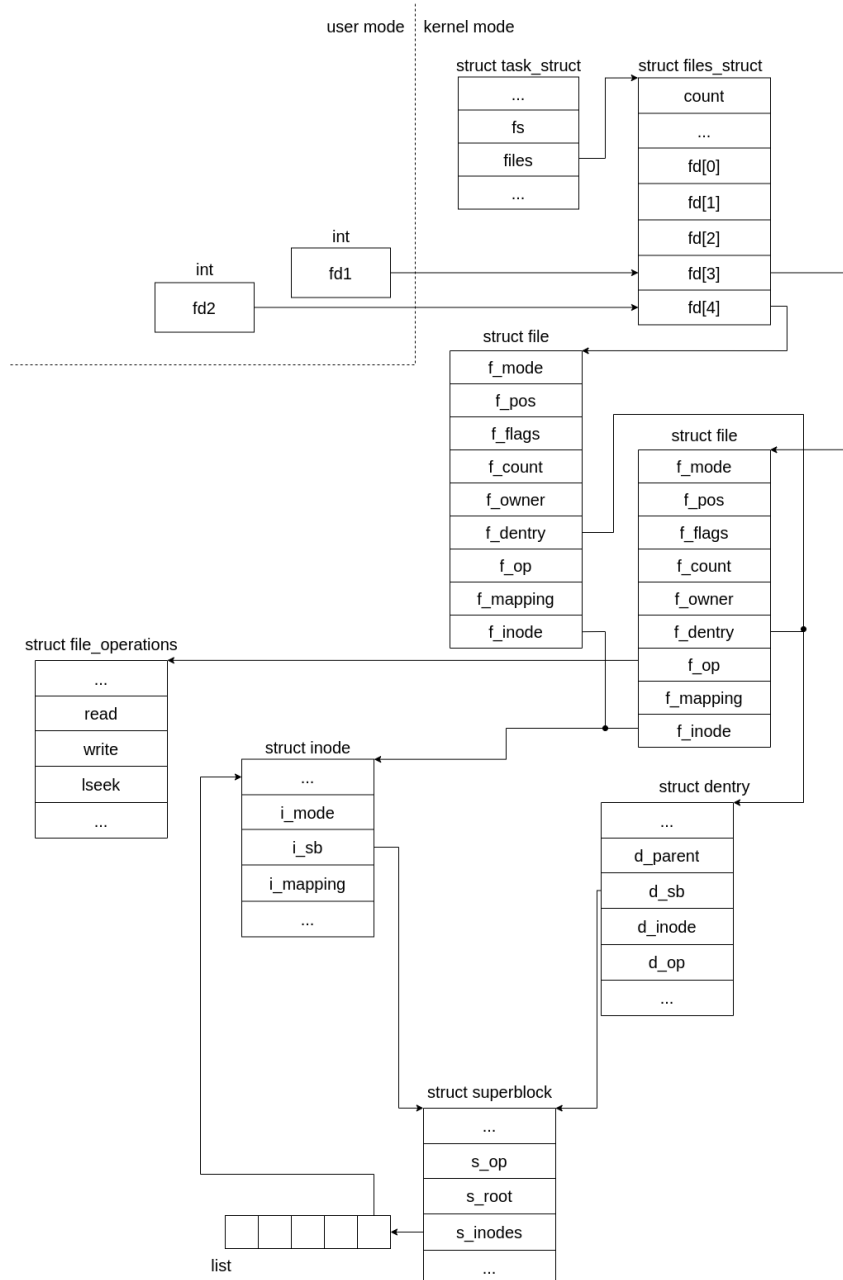


Рисунок 1.10 — Связь структур

Однопоточная реализация

```
1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <sys/stat.h>
```

```

4 #include <stdio.h>
5
6 #define PRINT_STAT(path, prefix) \
7     do { \
8         stat(path, &statbuf); \
9         fprintf(stdout, prefix ": inode num = %ld, size = %ld, \
10             blksize = %ld\n", \
11             statbuf.st_ino, statbuf.st_size, \
12             statbuf.st_blksize); \
13     } while (0)
14
15 int main() {
16     struct stat statbuf;
17     int file1 = open("output.txt", O_RDWR);
18     PRINT_STAT("output.txt", "open file1");
19     int file2 = open("output.txt", O_RDWR);
20     PRINT_STAT("output.txt", "open file2");
21
22     for (char ch = 'a'; ch <= 'z'; ch++) {
23         if (ch % 2)
24             write(file1, &ch, 1);
25         else
26             write(file2, &ch, 1);
27         PRINT_STAT("output.txt", "write");
28     }
29
30     close(file1);
31     PRINT_STAT("output.txt", "close file1");
32     close(file2);
33     PRINT_STAT("output.txt", "close file2");
34     return 0;
35 }

```

```

narandaev@hp ~/l/b/b/6/o/l/4 (main)> make start
./single.out
open file1: inode num = 6686618, size = 0, blksize = 4096
open file2: inode num = 6686618, size = 0, blksize = 4096
write: inode num = 6686618, size = 1, blksize = 4096
write: inode num = 6686618, size = 1, blksize = 4096
write: inode num = 6686618, size = 2, blksize = 4096
write: inode num = 6686618, size = 2, blksize = 4096
write: inode num = 6686618, size = 3, blksize = 4096
write: inode num = 6686618, size = 3, blksize = 4096
write: inode num = 6686618, size = 4, blksize = 4096
write: inode num = 6686618, size = 4, blksize = 4096
write: inode num = 6686618, size = 5, blksize = 4096
write: inode num = 6686618, size = 5, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 7, blksize = 4096
write: inode num = 6686618, size = 7, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 9, blksize = 4096
write: inode num = 6686618, size = 9, blksize = 4096
write: inode num = 6686618, size = 10, blksize = 4096
write: inode num = 6686618, size = 10, blksize = 4096
write: inode num = 6686618, size = 11, blksize = 4096
write: inode num = 6686618, size = 11, blksize = 4096
write: inode num = 6686618, size = 12, blksize = 4096
write: inode num = 6686618, size = 12, blksize = 4096
write: inode num = 6686618, size = 13, blksize = 4096
write: inode num = 6686618, size = 13, blksize = 4096
close file1: inode num = 6686618, size = 13, blksize = 4096
close file2: inode num = 6686618, size = 13, blksize = 4096
narandaev@hp ~/l/b/b/6/o/l/4 (main)> cat ./output.txt
bdfhjlnprtvxz

```

Рисунок 1.11 — Вывод программы

В программе файл дважды открывается на запись функцией `open()`. В системной таблице открытых файлов создаётся два дескриптора `struct file`, каждый из которых имеет собственный указатель `f_pos`, но оба ссылаются на один и тот же `inode`. С помощью системного вызова `write()` выполняется небуферизованный вывод. При изменении порядка вызова функций `close()` вывод программы не изменяется, так как вывод не буферизуется.

```

narandaev@hp ~/l/b/b/6/o/l/4 (main)> make start
./single.out
open file1: inode num = 6686618, size = 0, blksize = 4096
open file2: inode num = 6686618, size = 0, blksize = 4096
write: inode num = 6686618, size = 1, blksize = 4096
write: inode num = 6686618, size = 2, blksize = 4096
write: inode num = 6686618, size = 3, blksize = 4096
write: inode num = 6686618, size = 4, blksize = 4096
write: inode num = 6686618, size = 5, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 7, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 9, blksize = 4096
write: inode num = 6686618, size = 10, blksize = 4096
write: inode num = 6686618, size = 11, blksize = 4096
write: inode num = 6686618, size = 12, blksize = 4096
write: inode num = 6686618, size = 13, blksize = 4096
write: inode num = 6686618, size = 14, blksize = 4096
write: inode num = 6686618, size = 15, blksize = 4096
write: inode num = 6686618, size = 16, blksize = 4096
write: inode num = 6686618, size = 17, blksize = 4096
write: inode num = 6686618, size = 18, blksize = 4096
write: inode num = 6686618, size = 19, blksize = 4096
write: inode num = 6686618, size = 20, blksize = 4096
write: inode num = 6686618, size = 21, blksize = 4096
write: inode num = 6686618, size = 22, blksize = 4096
write: inode num = 6686618, size = 23, blksize = 4096
write: inode num = 6686618, size = 24, blksize = 4096
write: inode num = 6686618, size = 25, blksize = 4096
write: inode num = 6686618, size = 26, blksize = 4096
close file1: inode num = 6686618, size = 26, blksize = 4096
close file2: inode num = 6686618, size = 26, blksize = 4096
narandaev@hp ~/l/b/b/6/o/l/4 (main)> cat ./output.txt
abcdefghijklmnopqrstuvwxyz

```

Рисунок 1.12 — Вывод программы с использованием флага O_APPEND

Чтобы вывести алфавит полностью, можно использовать системный вызов `open()` с флагом `O_APPEND`. В таком случае будет происходить дозапись всех символов.

Многопоточная реализация

```

1 #include <fcntl.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #include <stdio.h>
5 #include <sys/stat.h>
6
7 #define PRINT_STAT(path, prefix) \
8     do { \
9         stat(path, &statbuf); \

```



```

10     fprintf(stdout, prefix ": inode num = %ld, size = %ld,
        blksize = %ld\n", \
11     statbuf.st_ino, statbuf.st_size, \
12     statbuf.st_blksize); \
13 } while (0)
14
15 void *task(void *payload){
16     int n = (int)payload;
17     int file = open("output.txt", O_WRONLY);
18     struct stat statbuf;
19
20     for (char ch = 'a'; ch <= 'z'; ch++)
21     if (ch % 2 == n) {
22         write(file, &ch, 1);
23         PRINT_STAT("output.txt", "write");
24     }
25     close(file);
26     return 0;
27 }
28
29 int main() {
30     pthread_t thids[2];
31     pthread_create(thids, NULL, task, (void *)0);
32     pthread_create(thids+1, NULL, task, (void *)1);
33     pthread_join(thids[0], NULL);
34     pthread_join(thids[1], NULL);
35     return 0;
36 }

```

```

narandaev@hp ~/l/b/b/6/o/l/4 (main)> make start-m
./multithread.out
write: inode num = 6686618, size = 1, blksize = 4096
write: inode num = 6686618, size = 2, blksize = 4096
write: inode num = 6686618, size = 3, blksize = 4096
write: inode num = 6686618, size = 4, blksize = 4096
write: inode num = 6686618, size = 1, blksize = 4096
write: inode num = 6686618, size = 5, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 7, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 9, blksize = 4096
write: inode num = 6686618, size = 10, blksize = 4096
write: inode num = 6686618, size = 9, blksize = 4096
write: inode num = 6686618, size = 11, blksize = 4096
write: inode num = 6686618, size = 11, blksize = 4096
write: inode num = 6686618, size = 12, blksize = 4096
write: inode num = 6686618, size = 13, blksize = 4096
write: inode num = 6686618, size = 12, blksize = 4096
write: inode num = 6686618, size = 13, blksize = 4096
write: inode num = 6686618, size = 13, blksize = 4096
narandaev@hp ~/l/b/b/6/o/l/4 (main)> cat ./output.txt
acegiknpqtvxz

```

Рисунок 1.13 — Вывод программы

Поведение данной программы подобно поведению многопоточной программы с использованием вызовов из `stdio`.

```

narandaev@hp ~/l/b/b/6/o/l/4 (main)> make start-m
./multithread.out
write: inode num = 6686618, size = 2, blksize = 4096
write: inode num = 6686618, size = 3, blksize = 4096
write: inode num = 6686618, size = 2, blksize = 4096
write: inode num = 6686618, size = 4, blksize = 4096
write: inode num = 6686618, size = 5, blksize = 4096
write: inode num = 6686618, size = 6, blksize = 4096
write: inode num = 6686618, size = 7, blksize = 4096
write: inode num = 6686618, size = 8, blksize = 4096
write: inode num = 6686618, size = 10, blksize = 4096
write: inode num = 6686618, size = 10, blksize = 4096
write: inode num = 6686618, size = 11, blksize = 4096
write: inode num = 6686618, size = 12, blksize = 4096
write: inode num = 6686618, size = 14, blksize = 4096
write: inode num = 6686618, size = 14, blksize = 4096
write: inode num = 6686618, size = 15, blksize = 4096
write: inode num = 6686618, size = 16, blksize = 4096
write: inode num = 6686618, size = 17, blksize = 4096
write: inode num = 6686618, size = 18, blksize = 4096
write: inode num = 6686618, size = 20, blksize = 4096
write: inode num = 6686618, size = 21, blksize = 4096
write: inode num = 6686618, size = 21, blksize = 4096
write: inode num = 6686618, size = 23, blksize = 4096
write: inode num = 6686618, size = 24, blksize = 4096
write: inode num = 6686618, size = 22, blksize = 4096
write: inode num = 6686618, size = 25, blksize = 4096
write: inode num = 6686618, size = 26, blksize = 4096
narandaev@hp ~/l/b/b/6/o/l/4 (main)> cat ./output.txt
bacedgfi hkjmlonqpsrutwvxzy
narandaev@hp ~/l/b/b/6/o/l/4 (main)>

```

Рисунок 1.14 — Вывод программы с использованием флага O_APPEND

В многопоточной реализации с использованием флага O_APPEND в файл будет записан весь алфавит, однако порядок символов не определён из-за параллельного выполнения потоков.

2. Структуры ядра

2.1. `_IO_FILE`

```
1 struct _IO_FILE {
2     int _flags;
3 #define _IO_file_flags _flags
4     char* _IO_read_ptr; /* Current read pointer */
5     char* _IO_read_end; /* End of get area. */
6     char* _IO_read_base; /* Start of putback+get area. */
7     char* _IO_write_base; /* Start of put area. */
8     char* _IO_write_ptr; /* Current put pointer. */
9     char* _IO_write_end; /* End of put area. */
10    char* _IO_buf_base; /* Start of reserve area. */
11    char* _IO_buf_end; /* End of reserve area. */
12    /* The following fields are used to support backing up and undo
13     . */
13    char *_IO_save_base; /* Pointer to start of non-current get
14        area. */
14    char *_IO_backup_base; /* Pointer to first valid character of
15        backup area */
15    char *_IO_save_end; /* Pointer to end of non-current get area.
16        */
16    struct _IO_marker *_markers;
17    struct _IO_FILE *_chain;
18    int _fileno;
19 #if 0
20    int _blksize;
21 #else
22    int _flags2;
23 #endif
24    _IO_off_t _old_offset; /* This used to be _offset but it's too
19    small. */
```

```

25
26 #define __HAVE_COLUMN /* temporary */
27 /* 1+column number of pbase(); 0 is unknown. */
28 unsigned short _cur_column;
29 signed char _vtable_offset;
30 char _shortbuf[1];
31 /* char* _save_gp_ptr; char* _save_eg_ptr; */
32 _IO_lock_t *_lock;
33 #ifdef _IO_USE_OLD_IO_FILE
34 };

```

2.2. files_struct

```

1 /*
2  * Open file table structure
3  */
4 struct files_struct {
5 /*
6  * read mostly part
7  */
8 atomic_t count;
9 bool resize_in_progress;
10 wait_queue_head_t resize_wait;
11 struct fdtable __rcu *fdt;
12 struct fdtable fdtab;
13 /*
14  * written part on a separate cache line in SMP
15  */
16 spinlock_t file_lock ____cacheline_aligned_in_smp;
17 unsigned int next_fd;
18 unsigned long close_on_exec_init[1];
19 unsigned long open_fds_init[1];
20 unsigned long full_fds_bits_init[1];
21 struct file __rcu * fd_array[NR_OPEN_DEFAULT];
22 };

```

2.3. fdtable

```
1 struct fdtable {
2     unsigned int max_fds;
3     struct file __rcu **fd;      /* current fd array */
4     unsigned long *close_on_exec;
5     unsigned long *open_fds;
6     unsigned long *full_fds_bits;
7     struct rcu_head rcu;
8 };
```

2.4. file

```
1 struct file {
2     union {
3         struct llist_node f_llist;
4         struct rcu_head f_rcuhead;
5         unsigned int f_iocb_flags;
6     };
7     struct path f_path;
8     struct inode *f_inode; /* cached value */
9     const struct file_operations *f_op;
10    /*
11     * Protects f_ep, f_flags.
12     * Must not be taken from IRQ context.
13     */
14     spinlock_t f_lock;
15     atomic_long_t f_count;
16     unsigned int f_flags;
17     fmode_t f_mode;
18     struct mutex f_pos_lock;
19     loff_t f_pos;
20     struct fown_struct f_owner;
21     const struct cred *f_cred;
22     struct file_ra_state f_ra;
```

```

23     u64         f_version;
24 #ifdef CONFIG_SECURITY
25     void         *f_security;
26 #endif
27     /* needed for tty driver, and maybe others */
28     void         *private_data;
29 #ifdef CONFIG_EPOLL
30     /* Used by fs/eventpoll.c to link all the hooks to this file */
31     struct hlist_head *f_ep;
32 #endif /* #ifdef CONFIG_EPOLL */
33     struct address_space *f_mapping;
34     errseq_t      f_wb_err;
35     errseq_t      f_sb_err; /* for syncfs */
36 } __randomize_layout
37 __attribute__((aligned(4))); /* lest something weird decides
    that 2 is OK */

```

2.5. inode

```

1 struct inode {
2     umode_t      i_mode;
3     unsigned short i_opflags;
4     kuid_t       i_uid;
5     kgid_t       i_gid;
6     unsigned int  i_flags;
7
8 #ifdef CONFIG_FS_POSIX_ACL
9     struct posix_acl *i_acl;
10    struct posix_acl *i_default_acl;
11 #endif
12
13    const struct inode_operations *i_op;
14    struct super_block *i_sb;
15    struct address_space *i_mapping;
16

```

```

17 #ifdef CONFIG_SECURITY
18     void          *i_security;
19 #endif
20
21     /* Stat data, not accessed from path walking */
22     unsigned long   i_ino;
23     /*
24      * Filesystems may only read i_nlink directly. They shall use
25      the
26      * following functions for modification:
27      *
28      *      (set/clear/inc/drop)_nlink
29      *      inode_(inc/dec)_link_count
30      */
31     union {
32         const unsigned int i_nlink;
33         unsigned int __i_nlink;
34     };
35     dev_t          i_rdev;
36     loff_t          i_size;
37     struct timespec64 i_atime;
38     struct timespec64 i_mtime;
39     struct timespec64 i_ctime;
40     spinlock_t      i_lock; /* i_blocks, i_bytes, maybe i_size */
41     unsigned short   i_bytes;
42     u8               i_blkbits;
43     u8               i_write_hint;
44     blkcnt_t         i_blocks;
45
46 #ifdef __NEED_I_SIZE_ORDERED
47     seqcount_t        i_size_seqcount;
48 #endif
49
50     /* Misc */
51     unsigned long     i_state;
52     struct rw_semaphore i_rwsem;

```



```

52
53     unsigned long    dirtied_when; /* jiffies of first dirtying */
54     unsigned long    dirtied_time_when;
55
56     struct hlist_node i_hash;
57     struct list_head i_io_list; /* backing dev IO list */
58 #ifdef CONFIG_CGROUP_WRITEBACK
59     struct bdi_writeback *i_wb; /* the associated cgroup wb */
60
61     /* foreign inode detection, see wbc_detach_inode() */
62     int      i_wb_frn_winner;
63     u16      i_wb_frn_avg_time;
64     u16      i_wb_frn_history;
65 #endif
66     struct list_head i_lru; /* inode LRU list */
67     struct list_head i_sb_list;
68     struct list_head i_wb_list; /* backing dev writeback list */
69     union {
70         struct hlist_head i_dentry;
71         struct rcu_head i_rcu;
72     };
73     atomic64_t i_version;
74     atomic64_t i_sequence; /* see futex */
75     atomic_t i_count;
76     atomic_t i_dio_count;
77     atomic_t i_writecount;
78 #if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
79     atomic_t i_readcount; /* struct files open RO */
80 #endif
81     union {
82         const struct file_operations *i_fop; /* former ->i_op->
            default_file_ops */
83         void (*free_inode)(struct inode *);
84     };
85     struct file_lock_context *i_flctx;
86     struct address_space i_data;

```

```

87     struct list_head    i_devices;
88     union {
89         struct pipe_inode_info    *i_pipe;
90         struct cdev    *i_cdev;
91         char    *i_link;
92         unsigned    i_dir_seq;
93     };
94
95     __u32    i_generation;
96
97 #ifdef CONFIG_FSNOTIFY
98     __u32    i_fsnotify_mask; /* all events this inode cares about
99                                */
100     struct fsnotify_mark_connector __rcu    *i_fsnotify_marks;
101 #endif
102
103 #ifdef CONFIG_FS_ENCRYPTION
104     struct fscrypt_info *i_crypt_info;
105 #endif
106
107 #ifdef CONFIG_FS_VERITY
108     struct fsverity_info    *i_verity_info;
109 #endif
110
111     void    *i_private; /* fs or device private pointer */
112 } __randomize_layout;

```

2.6. dentry

```

1 struct dentry {
2     /* RCU lookup touched fields */
3     unsigned int d_flags;    /* protected by d_lock */
4     seqcount_spinlock_t d_seq; /* per dentry seqlock */
5     struct hlist_bl_node d_hash; /* lookup hash list */
6     struct dentry *d_parent; /* parent directory */

```

```

7  struct qstr d_name;
8  struct inode *d_inode;    /* Where the name belongs to - NULL
    is
9      * negative */
10 unsigned char d_iname[DNAME_INLINE_LEN]; /* small names */
11 /* Ref lookup also touches following */
12 struct lockref d_lockref; /* per-dentry lock and refcount */
13 const struct dentry_operations *d_op;
14 struct super_block *d_sb; /* The root of the dentry tree */
15 unsigned long d_time;    /* used by d_revalidate */
16 void *d_fsdata;    /* fs-specific data */
17 union {
18     struct list_head d_lru;    /* LRU list */
19     wait_queue_head_t *d_wait; /* in-lookup ones only */
20 };
21 struct list_head d_child; /* child of parent list */
22 struct list_head d_subdirs; /* our children */
23 /*
24  * d_alias and d_rcu can share memory
25  */
26 union {
27     struct hlist_node d_alias; /* inode alias list */
28     struct hlist_bl_node d_in_lookup_hash; /* only for in-lookup
        ones */
29     struct rcu_head d_rcu;
30 } d_u;
31 } __randomize_layout;

```

2.7. superblock

```

1 struct super_block {
2     struct list_head s_list;    /* Keep this first */
3     dev_t s_dev;    /* search index; _not_ kdev_t */
4     unsigned char s_blocksize_bits;
5     unsigned long s_blocksize;

```

```

6   loff_t      s_maxbytes; /* Max file size */
7   struct file_system_type *s_type;
8   const struct super_operations *s_op;
9   const struct dquot_operations *dq_op;
10  const struct quotactl_ops *s_qcop;
11  const struct export_operations *s_export_op;
12  unsigned long s_flags;
13  unsigned long s_iflags; /* internal SB_I_* flags */
14  unsigned long s_magic;
15  struct dentry *s_root;
16  struct rw_semaphore s_umount;
17  int s_count;
18  atomic_t s_active;
19 #ifdef CONFIG_SECURITY
20     void *s_security;
21 #endif
22     const struct xattr_handler **s_xattr;
23 #ifdef CONFIG_FS_ENCRYPTION
24     const struct fscrypt_operations *s_cop;
25     struct fscrypt_keyring *s_master_keys; /* master crypto keys
        in use */
26 #endif
27 #ifdef CONFIG_FS_VERITY
28     const struct fsverity_operations *s_vop;
29 #endif
30 #ifdef CONFIG_UNICODE
31     struct unicode_map *s_encoding;
32     __u16 s_encoding_flags;
33 #endif
34     struct hlist_bl_head s_roots; /* alternate root dentries for
        NFS */
35     struct list_head s_mounts; /* list of mounts; _not_ for fs use
        */
36     struct block_device *s_bdev;
37     struct backing_dev_info *s_bdi;
38     struct mtd_info *s_mtd;

```

```

39  struct hlist_node s_instances;
40  unsigned int      s_quota_types;  /* Bitmask of supported quota
    types */
41  struct quota_info s_dquot;  /* Diskquota specific options */
42  struct sb_writers s_writers;
43  void      *s_fs_info; /* Filesystem private info */
44  u32      s_time_gran;
45  /* Time limits for c/m/atime in seconds */
46  time64_t      s_time_min;
47  time64_t      s_time_max;
48  #ifdef CONFIG_FSNOTIFY
49  __u32      s_fsnotify_mask;
50  struct fsnotify_mark_connector __rcu *s_fsnotify_marks;
51  #endif
52  char      s_id[32]; /* Informational name */
53  uuid_t      s_uuid;  /* UUID */
54  unsigned int      s_max_links;
55  fmode_t      s_mode;
56  struct mutex s_vfs_rename_mutex; /* Kludge */
57  const char *s_subtype;
58  const struct dentry_operations *s_d_op; /* default d_op for
    dentries */
59  int cleancache_poolid;
60  struct shrinker s_shrink; /* per-sb shrinker handle */
61  atomic_long_t s_remove_count;
62  atomic_long_t s_fsnotify_connectors;
63  int s_readonly_remount;
64  errseq_t s_wb_err;
65  struct workqueue_struct *s_dio_done_wq;
66  struct hlist_head s_pins;
67  struct user_namespace *s_user_ns;
68  struct list_lru      s_dentry_lru;
69  struct list_lru      s_inode_lru;
70  struct rcu_head      rcu;
71  struct work_struct      destroy_work;
72  struct mutex      s_sync_lock; /* sync serialisation lock */

```

```

73  int s_stack_depth;
74  spinlock_t    s_inode_list_lock    ___cacheline_aligned_in_smp;
75  struct list_head s_inodes; /* all inodes */
76  spinlock_t    s_inode_wblist_lock;
77  struct list_head s_inodes_wb; /* writeback inodes */
78 } __randomize_layout;

```

2.8. stat

```

1  struct stat {
2      dev_t      st_dev;      /* ID of device containing file */
3      ino_t      st_ino;      /* inode number */
4      mode_t     st_mode;     /* protection */
5      nlink_t    st_nlink;    /* number of hard links */
6      uid_t      st_uid;      /* user ID of owner */
7      gid_t      st_gid;      /* group ID of owner */
8      dev_t      st_rdev;     /* device ID (if special file) */
9      off_t      st_size;     /* total size, in bytes */
10     blksize_t   st_blksize;  /* blocksize for file system I/O */
11     blkcnt_t    st_blocks;   /* number of 512B blocks allocated */
12     time_t      st_atime;    /* time of last access */
13     time_t      st_mtime;    /* time of last modification */
14     time_t      st_ctime;    /* time of last status change */
15 };

```