Linux System Programming Part 3 - Filesystem and Files

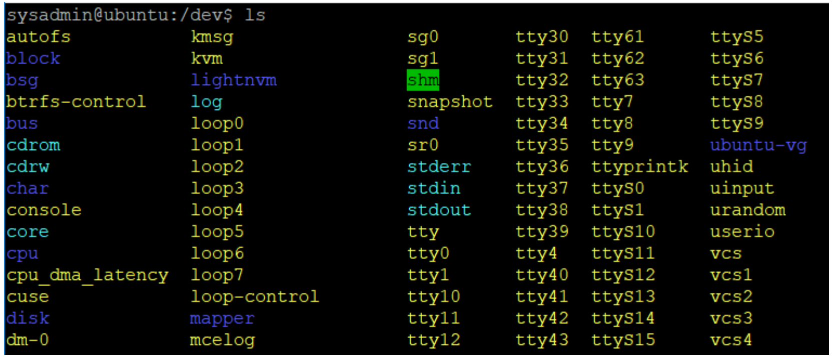
IBA Bulgaria
2021

Files in Linux

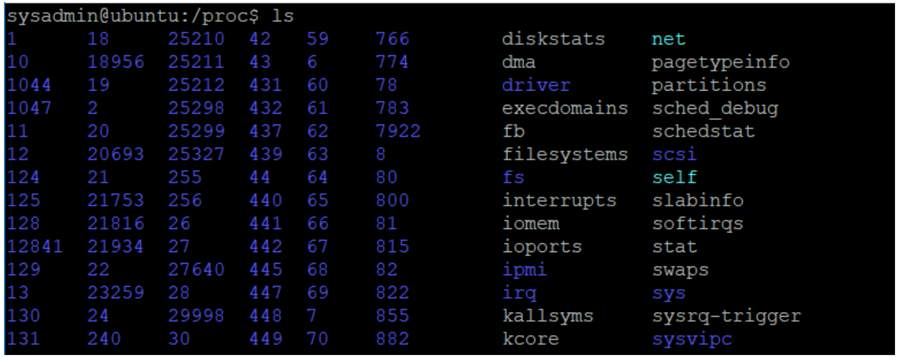
- “Everything is a file.”
- **inode -** data structure that describes a filesystem object.
- **Files** are always opened from user space by a **name**.
- A name and inode pair is called a **link**.
- **Regular files** - bytes of data.
- **Directories -** mapping between filenames and inodes (links).
- **Hard links** - multiple links map different names to the same inode.
- **Symbolic links -** like regular files which contain the complete pathname of the linked-to files.
- **Special files -** block device files, character device files, named pipes, Unix domain sockets.

Filesystems and namespaces

- Linux provides a **global and unified namespace** of files and directories (with a root ‘/’).
- A **filesystem** is a collection of files and directories in a formal and valid hierarchy.
- Filesystems may be individually added (**mounted**) to and removed (**unmounted**) from the global namespace of files and directories.
- Some special filesystems - ‘/dev’ and ‘/proc’.



```
sysadmin@ubuntu:/dev$ ls
autofs           kmsg                sg0       tty30  tty61       ttyS5
block            kvm                 sg1       tty31  tty62       ttyS6
bsg              lightnvm            shm       tty32  tty63       ttyS7
btrfs-control    log                 snapshot  tty33  tty7        ttyS8
bus              loop0               snd       tty34  tty8        ttyS9
cdrom            loop1               sr0       tty35  tty9        ubuntu-vg
cdrw             loop2               stderr    tty36  ttyprintk   uhid
char             loop3               stdin     tty37  ttyS0       uinput
console          loop4               stdout    tty38  ttyS1       urandom
core             loop5               tty       tty39  ttyS10      userio
cpu              loop6               tty0      tty4   ttyS11      vcs
cpu_dma_latency  loop7               tty1      tty40  ttyS12      vcs1
cuse             loop-control        tty10     tty41  ttyS13      vcs2
disk             mapper              tty11     tty42  ttyS14      vcs3
dm-0             mcelog              tty12     tty43  ttyS15      vcs4
```



```
sysadmin@ubuntu:/proc$ ls
1       18     25210  42   59   766         diskstats    net
10      18956  25211  43   6    774         dma          pagetypeinfo
1044    19     25212  431  60   78          driver       partitions
1047    2      25298  432  61   783         execdomains  sched_debug
11      20     25299  437  62   7922        fb           schedstat
12      20693  25327  439  63   8           filesystems  scsi
124     21     255    44   64   80          fs           self
125     21753  256    440  65   800         interrupts   slabinfo
128     21816  26     441  66   81          iomem        softirqs
12841   21934  27     442  67   815         ioports      stat
129     22     27640  445  68   82          ipmi         swaps
13      23259  28     447  69   822         irq          sys
130     24     29998  448  7    855         kallsyms     sysrq-trigger
131     240    30     449  70   882         kcore        sysvipc
```

Working with Files in C

- Before a file can be read from or written to, it must be **opened**.
- Each open instance of a file is given a unique **file descriptor (fd)**.
- File descriptors are represented by the C ***int*** type.
- A single file can be opened more than once, by a different or even the same process.
- **open**() / **fopen**()- opens file and returns its file descriptor.
- **read**() / **fread**() - reads data from a file.
- **write**() / **fwrite**() - writes data into a file.
- **fflush**() - flushes a stream.
- **close**() - unmaps the a file descriptor with the associated file.
- **lseek**() / **fseek**() - set the file position of a file descriptor to a given value.
- **fcntl**() - manipulate file descriptor, for example for locking.
- **errno** - number of last error.

Buffered vs unbuffered streams

- **Stream** is a representation of flow of data from one side to another e.g. from disk to memory and from memory to disk.
- **File** is a representation to store data on disk file. File uses streams to store and load data.
- **Buffer** is (often) used to hold stream data temporarily.
- Characters written to or read from an **unbuffered** stream are transmitted individually to or from the file as soon as possible.
- Characters written to or read from a **fully buffered** stream are transmitted to or from the file in blocks of arbitrary size.
- Characters written to a **line buffered** stream are transmitted to the file in blocks when a newline character is encountered.

Buffered vs unbuffered streams

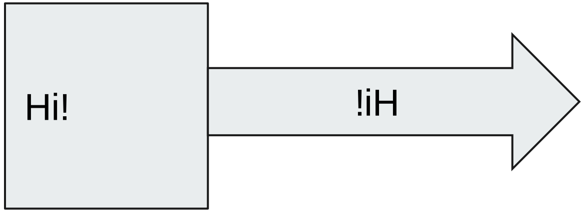
Low-level file routines (**unbuffered** streams):

- **open**(), **read**(), **write**(), **lseek**(), etc.
- Part of <**unistd.h**> library.
- Work with file descriptors of type **int**.
- Treat the input/output as **binary** data.

High-level file routines (**buffered** streams):

- **fopen**(), **fread**(), **fwrite**(), **fseek**(), etc.
- Part of <**stdio.h**> library.
- Use the object type **FILE**.
- Treat the input/output as **text** streams.
- The read/write of the accumulated buffer can be forced with **fflush**().

Unbuffered:

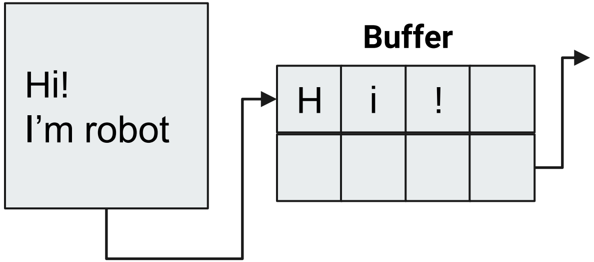


Hi!

!iH

Contents immediately available to the program

Buffered:



Hi!
I'm robot

Buffer

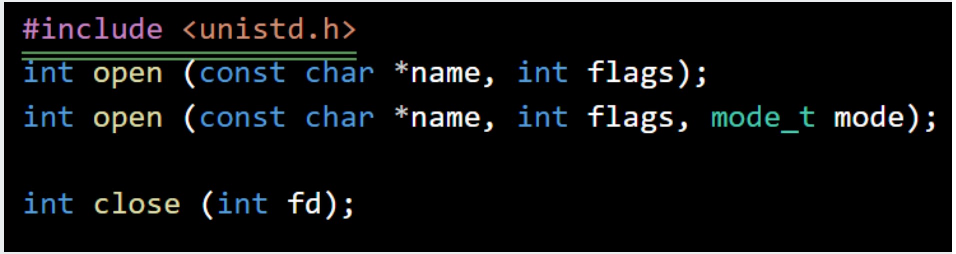
H	i	!	

Characters accumulated into a buffer

Buffer contents available to the program when:

- full buffer
- closed stream
- program terminates
- new line (*)
- flush

Basic file processing



```
#include <unistd.h>
int open (const char *name, int flags);
int open (const char *name, int flags, mode_t mode);

int close (int fd);
```

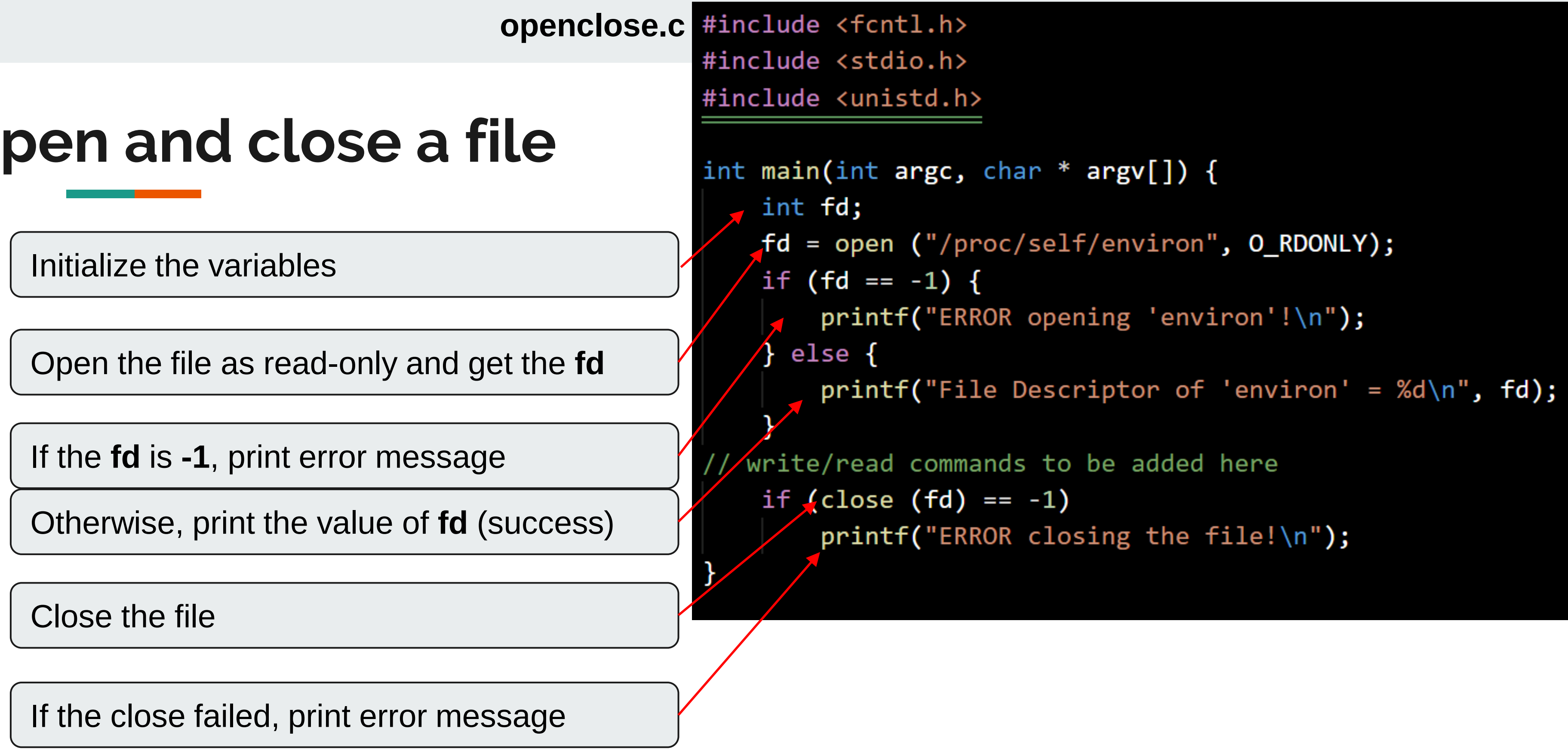
If **flags** has **O_CREAT** set, the **mode** is a bit mask of:

S_IRWXU	S_IWGRP
S_IRUSR	S_IXGRP
S_IWUSR	S_IRWXO
S_IXUSR	S_IROTH
S_IRWXG	S_IWOTH
S_IRGRP	S_IXOTH

flags parameter is a bit mask of the following bits:

O_APPEND	The file will be opened in append mode.
O_ASYNC	**SIGIO** generated when readable or writable.
O_CREAT	If the file doesn't exist - create it.
O_DIRECT	Opened for direct I/O.
O_DIRECTORY	If **name** is not a directory, **open**( ) will fail.
O_EXCL	If O_CREAT and file exists, **open**( ) will fail.
O_LARGEFILE	A file larger than 2G to be opened.
O_NOCTTY	This flag is not frequently used.
O_NOFOLLOW	If **name** is a symbolic link, **open**( ) will fail.
O_NONBLOCK	If possible, open in nonblocking mode.
O_SYNC	The file will be opened for synchronous I/O.
O_TRUNC	If the file exists, truncated it to zero length.

Open and close a file



openclose.c

Initialize the variables

Open the file as read-only and get the **fd**

If the **fd** is **-1**, print error message

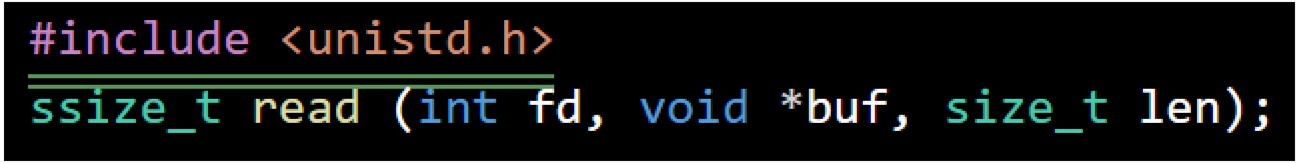
Otherwise, print the value of **fd** (success)

Close the file

If the close failed, print error message

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

int main(int argc, char * argv[]) {
    int fd;
    fd = open ("/proc/self/environ", O_RDONLY);
    if (fd == -1) {
        printf("ERROR opening 'environ'!\n");
    } else {
        printf("File Descriptor of 'environ' = %d\n", fd);
    }
// write/read commands to be added here
    if (close (fd) == -1)
        printf("ERROR closing the file!\n");
}
```

Reading file contents

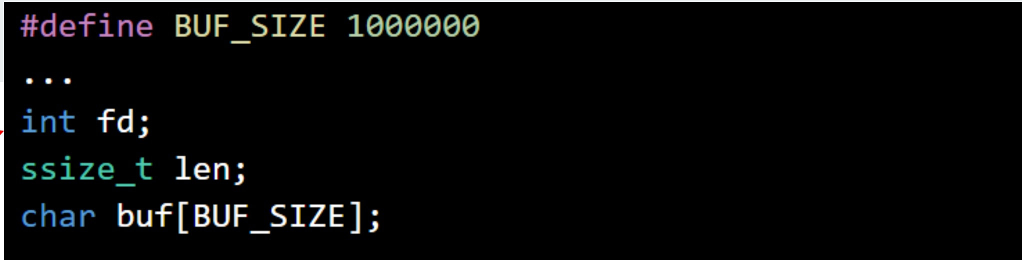


```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t len);
```

- Each call reads up to ***len*** bytes into ***buf*** from the current file offset of the file referenced by ***fd***.
- On success, the number of bytes written into ***buf*** is returned.
- On error, the call returns ***-1*** and ***errno*** is set.

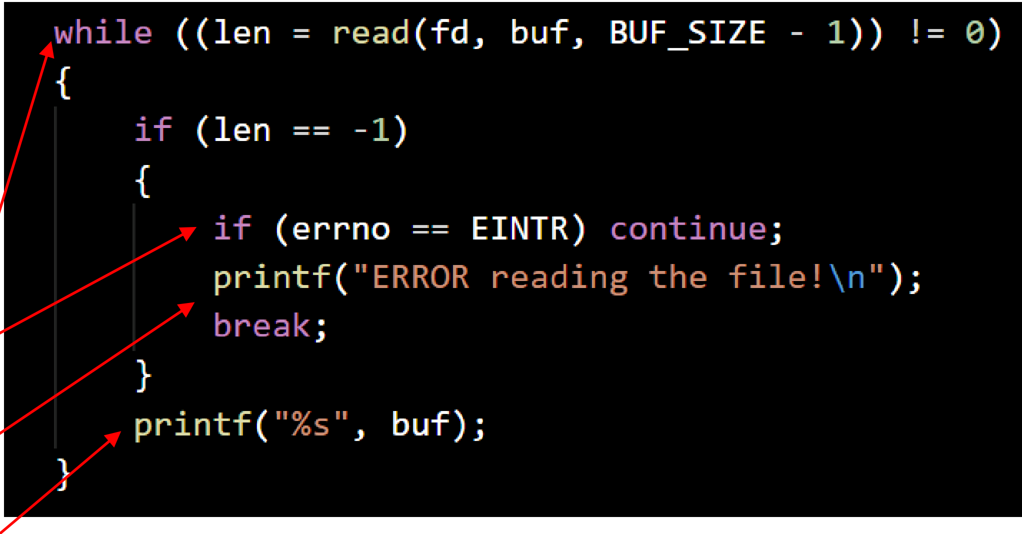
readfile.c

Read and print a file



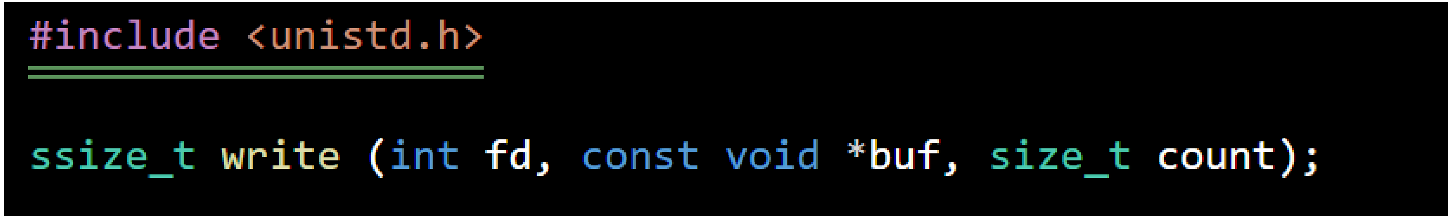
```
#define BUF_SIZE 1000000
...
int fd;
ssize_t len;
char buf[BUF_SIZE];
```

- Initialize the variables
- Open ‘**./readfile.c**’, if failed print error & exit
- While reading from file returns length **<> 0**
  - If length is -1 and errno is EINTR, try to read again
  - If length is -1 and errno <> EINTR, print error and finish reading
  - Otherwise print the buffer
- Close the file and notify on error



```
  while ((len = read(fd, buf, BUF_SIZE - 1)) != 0)
  {
      if (len == -1)
      {
          if (errno == EINTR) continue;
          printf("ERROR reading the file!\n");
          break;
      }
      printf("%s", buf);
  }
```

Create and write into a file



```
#include <unistd.h>

ssize_t write (int fd, const void *buf, size_t count);
```

- Writes up to ***count*** bytes starting at ***buf*** to the current file position of the file referenced by the file descriptor ***fd***.
- On success, the number of bytes written is returned, and the file position is updated.
- On error, the call returns ***-1*** and ***errno*** is set.

Write sentences into file

Initialize the variables

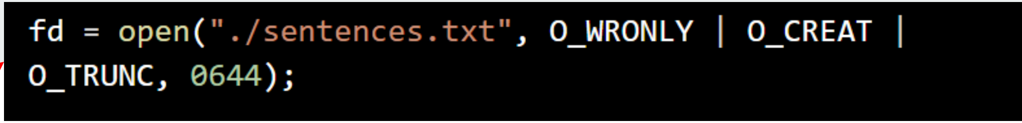
Create empty ‘**./sentences.txt**’, if failed print error & exit

Do 100 times

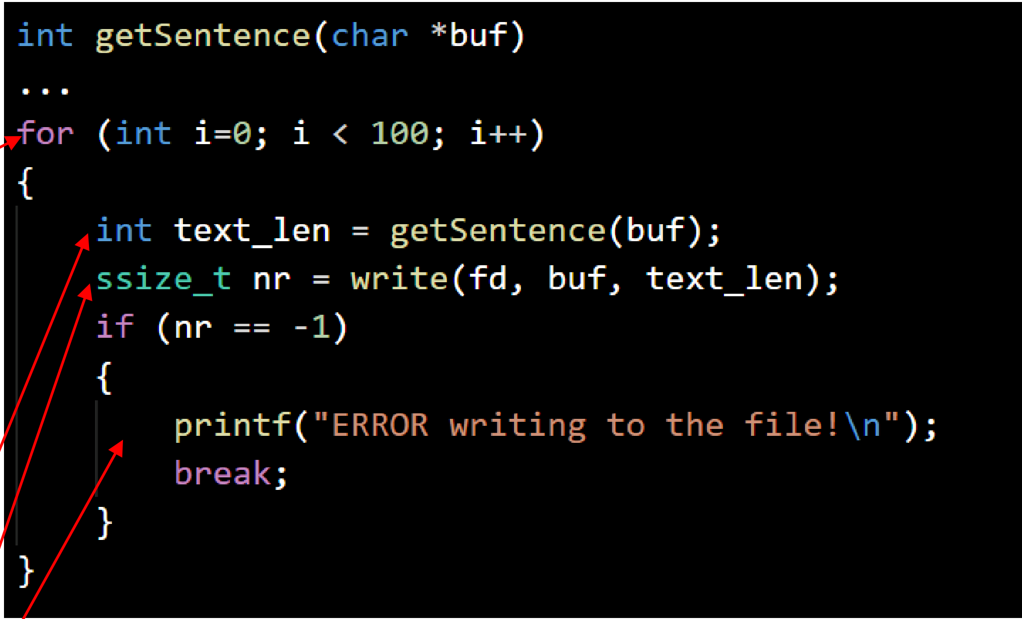
- Call **getSentence()** to get a new text into the buffer and its length
- Write the buffer into the file and get the number of bytes written
- If the number of written bytes is -1, print error and stop writing.

Close the file and notify the error or success

writefile.c

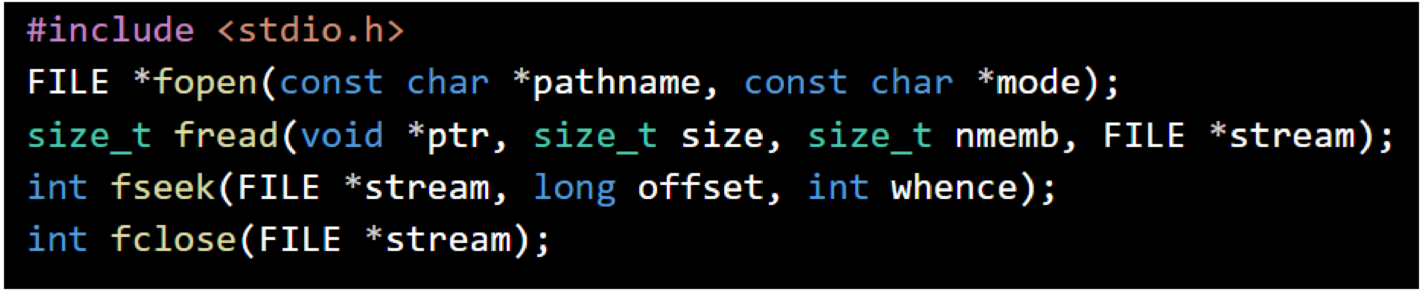


```c
fd = open("./sentences.txt", O_WRONLY | O_CREAT |
O_TRUNC, 0644);
```



```c
int getSentence(char *buf)
...
for (int i=0; i < 100; i++)
{
    int text_len = getSentence(buf);
    ssize_t nr = write(fd, buf, text_len);
    if (nr == -1)
    {
        printf("ERROR writing to the file!\n");
        break;
    }
}
```

Seeking in files and Sparse files



```c
#include <stdio.h>
FILE *fopen(const char *pathname, const char *mode);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
int fseek(FILE *stream, long offset, int whence);
int fclose(FILE *stream);
```

- The behavior of **fseek()** depends on the ***origin*** argument: **SEEK_CUR**, **SEEK_END** or **SEEK_SET**.
- The call returns the new file position on success.
- On error, the call returns ***-1*** and ***errno*** is set.
- Seeking after the end of a file and then writing in it causes **holes** padded with zeros.
- Files with holes are called **sparse files**.
- Holes do not occupy any physical disk space.
- **du** - estimate file space usage.

makesparse.c

Make a file with a hole

Initialize the variables

Create a file, its name is the first argument passed to the program

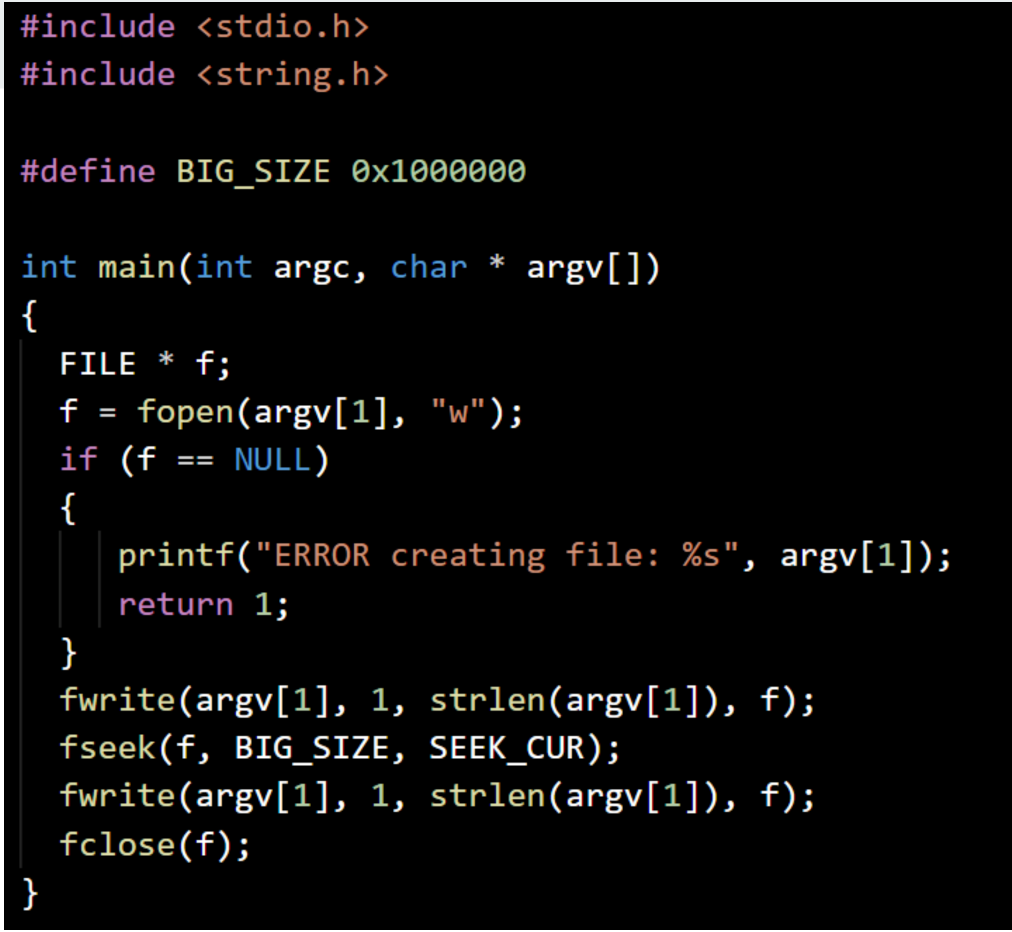
If failed to open - notify error and exit

Write the name of the file into the file

Jump 16777216 bytes forward into the file

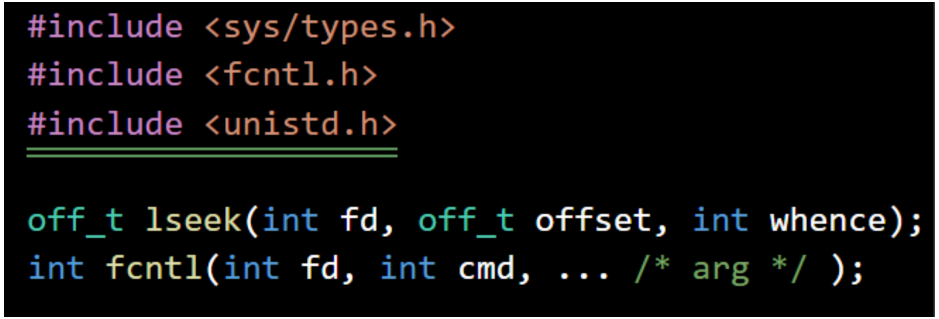
Write the name of the file into the file again

Close the file

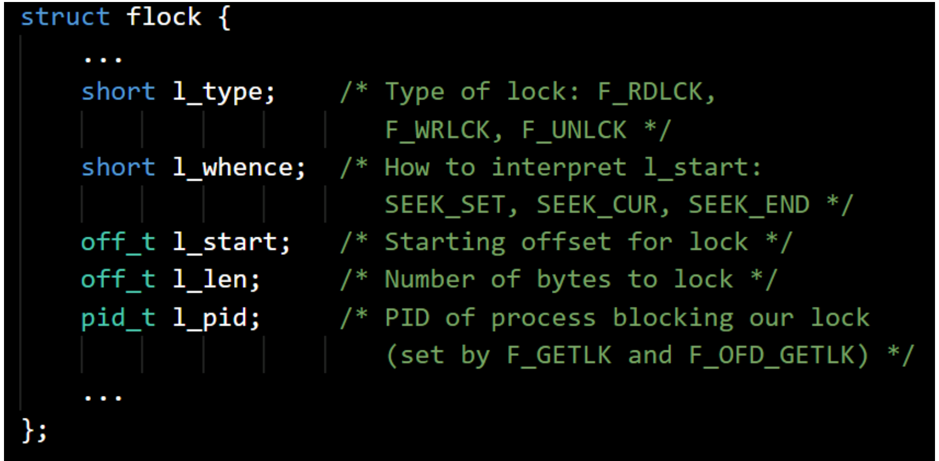


```c
#include <stdio.h>
#include <string.h>

#define BIG_SIZE 0x1000000

int main(int argc, char * argv[])
{
  FILE * f;
  f = fopen(argv[1], "w");
  if (f == NULL)
  {
    printf("ERROR creating file: %s", argv[1]);
    return 1;
  }
  fwrite(argv[1], 1, strlen(argv[1]), f);
  fseek(f, BIG_SIZE, SEEK_CUR);
  fwrite(argv[1], 1, strlen(argv[1]), f);
  fclose(f);
}
```

Locking files



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

off_t lseek(int fd, off_t offset, int whence);
int fcntl(int fd, int cmd, ... /* arg */ );
```



```c
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK,
                        F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start:
                        SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock
                        (set by F_GETLK and F_OFD_GETLK) */
    ...
};
```

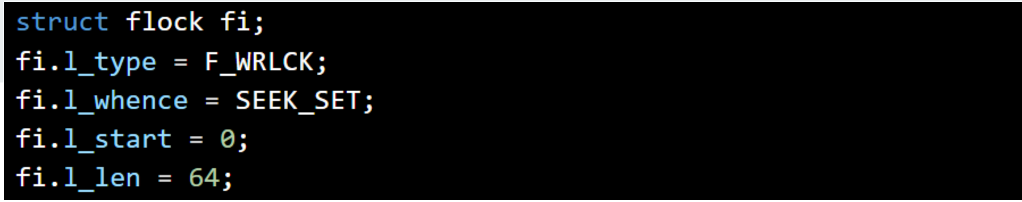
Use the **fcntl()** call providing a pointer to a **flock** structure. This call manipulates the file descriptor ***fd***, depending on the command ***cmd***:

- To lock a block of a file use **F_SETLK.**
- If the block is already locked use **F_GETLK** to get information about the locking process.
- To unlock a block of a file use **F_SETLK** but set **flock.l_type = F_UNLCK**.

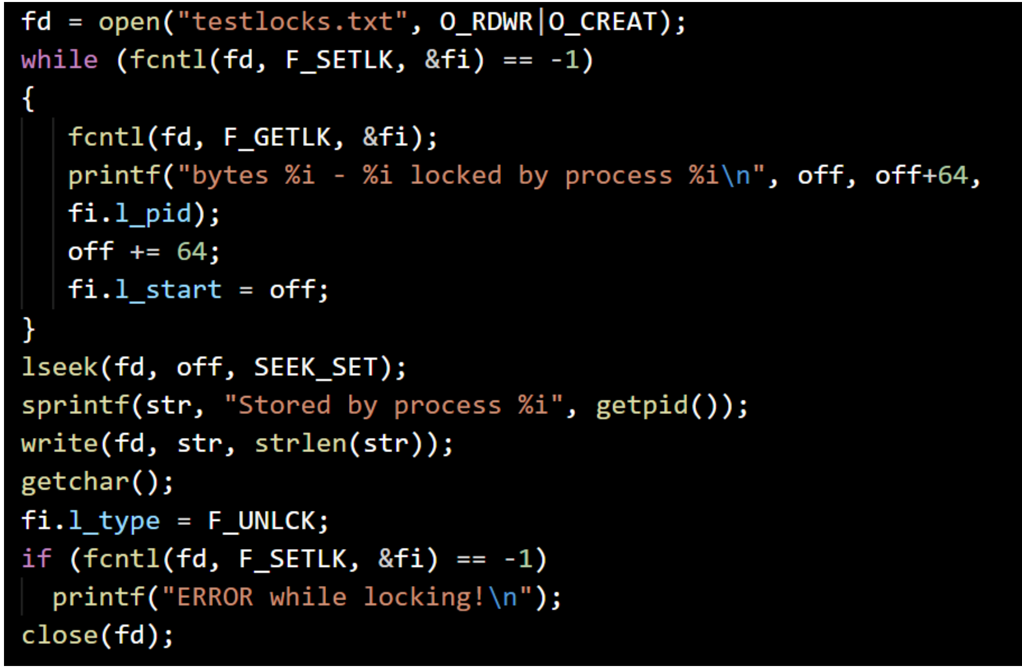
lockfile.c

Lock and write in there

- Initialize the variables
- Open for writing or create ‘**./testlocks.txt**’
- While lock of 64 bytes from **offset** fails:
    - Get information about the locking process and print info about it
    - Move the **offset** 64 bytes further
- Move **offset** bytes from the start of the file and write info about the current process
- Wait for **Enter** key press
- Unlock the 64 locked bytes, notify if failed
- Close the file



```
struct flock fi;
fi.l_type = F_WRLCK;
fi.l_whence = SEEK_SET;
fi.l_start = 0;
fi.l_len = 64;
```



```
fd = open("testlocks.txt", O_RDWR|O_CREAT);
while (fcntl(fd, F_SETLK, &fi) == -1)
{
    fcntl(fd, F_GETLK, &fi);
    printf("bytes %i - %i locked by process %i\n", off, off+64,
    fi.l_pid);
    off += 64;
    fi.l_start = off;
}
lseek(fd, off, SEEK_SET);
sprintf(str, "Stored by process %i", getpid());
write(fd, str, strlen(str));
getchar();
fi.l_type = F_UNLCK;
if (fcntl(fd, F_SETLK, &fi) == -1)
  printf("ERROR while locking!\n");
close(fd);
```

Exercise

Program *FileManipulator*:

Write a program (‘**fmanipulator.c**’), which takes 3 arguments: **words_count**, **min_length**, **max_length**. The program should generate a file called ‘**words.txt**’, which contains **words_count** words (separated with spaces). To generate the words use the following function from the code files in ‘**/day03/rndword/**’ (note that the function will not generate real words, but rather random sequences of characters):

```c
char * rndword(int min_length, int max_length);
```

Then the program should print the generated file to the screen.

After the program creates ‘**words.txt**’ and before writing into it it should lock the first 100 bytes of the file. When the program prints out the generated words it should wait for **Enter** key press and then unlock the file. Respectively, if the file was already locked by another process, the program should notify about this and wait for **Enter** key press, before trying again (to lock, generate and write).

Compile and run a couple of instances to test.