Introduction to System Calls and I/O

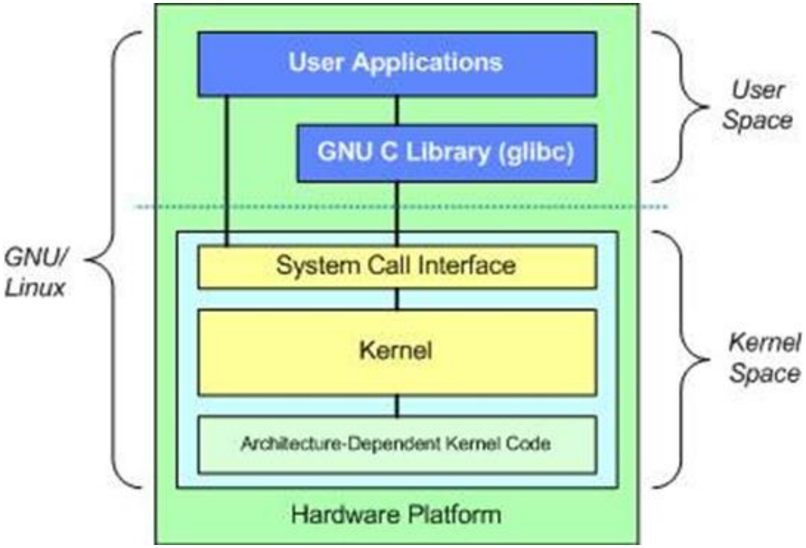
Kaynak: http://web.eecs.utk.edu/~jplank/plank/classes/cs360/360/notes/Syscall-Intro/lecture.html

CS360 Lecture notes James S. Plank

System Calls

- The way that programs talk to the operating system is via ``*system calls*.''
- A system call looks like a procedure call but it's different -- **it is a request to the operating system to perform some activity**.
- System calls are expensive. A procedure call can usually be performed in a few machine instructions.
- A system call requires the computer to save its state, let the operating system take control of the CPU, have the operating system perform some function, have the operating system save its state, and then have the operating system give control of the CPU back to you.
- Some examples:
  - signal()
  - getpid()
  - kill()
  - stat()
  - fork()
  - read()

  …



Source: http://www.ibm.com/developerworks/library/l-linux-kernel/

I/O (Giriş/Çıkış, Input/Output)

- It is important for security that the operating system controls I / O processes.
- When does I / O occur?
  - Through the keyboard and screen between the program and the user.
  - Between the program and the file storage system
  - Between the program and a hardware or peripheral unit.
- I/O system calls:
  - open()
  - close()
  - read()
  - write()
  - lseek()
- Library calls: fopen,fwrite, fclose, fseek,scanf, putc, etc..

System Calls for I/O

- I/O system calls:
  - int **open**(char *path, int flags [, int mode]);
  - int **close**(int fd);
  - ssize_t **read**(int fd, void *buf, size_t count);
  - ssize_t **write**(int fd, const void *buf, size_t count);
  - off_t **lseek**(int fd, off_t offset, int whence);
- ssize_t and off_t are ints and longs
- You'll note that they look like regular procedure calls. This is how you program with them -- like regular procedure calls.
- However, you should know that they are different: A system call makes a request to the operating system.
- A procedure call just jumps to a procedure defined elsewhere in your program. That procedure call may itself make a system call (for example, fopen() calls open()).

Open

- int **open**(char *path, int flags [, int mode]);

- If the operating system approves your request, it will return a ``file descriptor'' to you.
- This is a non-negative integer. If it returns -1, then you have been denied access, and you have to check the value of the variable "errno" to determine why. (That or use perror()).

- All actions that you will perform on files will be done through the operating system.
- Whenever you want to do file I/O, you specify the file by its file descriptor. You must first open that file to get a file descriptor.

- Open makes a request to the operating system to use a file.
- Path: specifies what file you would like to use.
- Flags, Mode: specify how you would like to use the file

Open - flags

- include: <fcntl.h>
- **flags** = bitwise | or of any of the following:

O_RDONLY	Only read operations permitted
O_WRONLY	Only write operations permitted
O_RDWR	Read and Write operations both permitted
O_APPEND	All writes go to end of file
O_CREAT	Create file if it doesn't already exist
O_TRUNC	Delete existing contents of file
O_EXCL	Open fails if file already exists
O_SHLOCK	Get a "shared lock" on the file
O_EXLOCK	Get an "exclusive lock" on the file
O_NONBLOCK	Non-blocking, applies to open operation only
O_DIRECT	Try to avoid all caching of operations
O_FSYNC	All writes immediately effective, no buffering
O_NOFOLLOW	If file is symbolic link, open it, don't follow it

Open - modes

- There are also highly un-memorable named constants if <sys/stat.h> is included,
- they may be added together in any combination:

S_IRUSR	Owner may read
S_IWUSR	Owner may write
S_IXUSR	Owner may execute
S_IRGRP	Members of group may read
S_IWGRP	Members of group may write
S_IXGRP	Members of group may execute
S_IROTH	Others may read
S_IWOTH	Others may write
S_IXOTH	Others may execute

Open

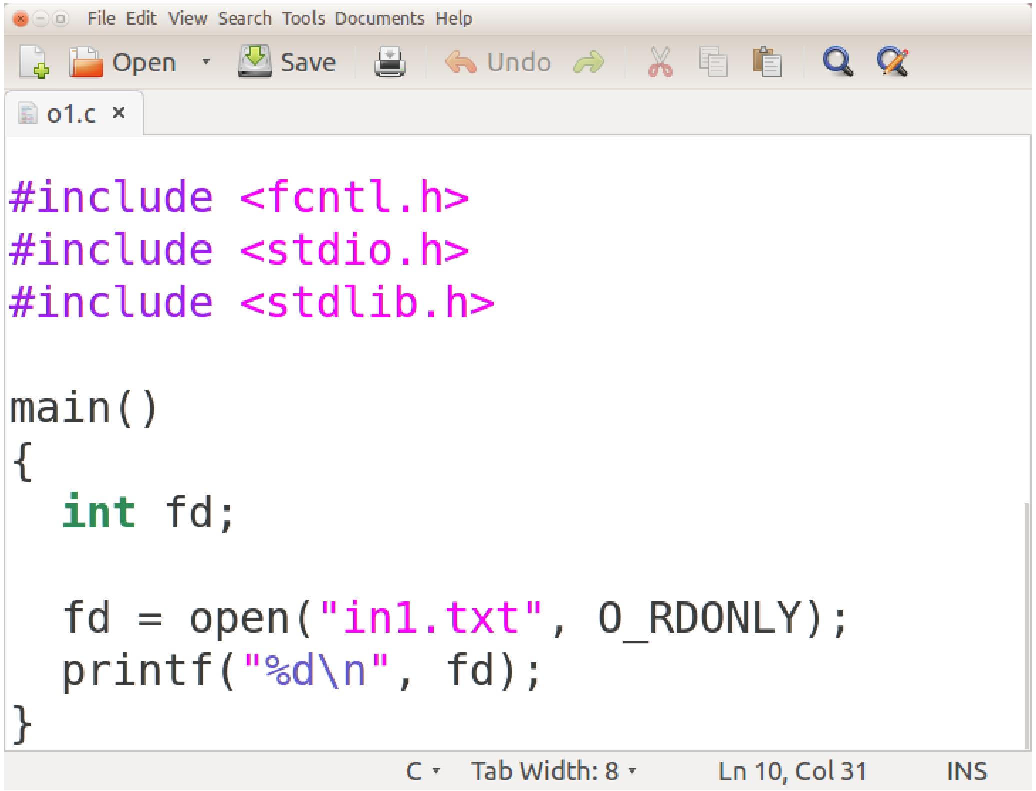
- **mode** is required if file is created, ignored otherwise.
- mode specifies the protection bits, e.g. 0644 = rw-r--r--
- 0 (octal notation) –User-Group- Other
- 0 6 4 4

0644 is the most typical value -- it says "I can read and write it; everyone else can only read it.
0755 :"I can read, write and execute it; and everyone else can only read and execute it.

rwx	oct	meaning
---	---	-------
001	01	= execute
010	02	= write
011	03	= write & execute
100	04	= read
101	05	= read & execute
110	06	= read & write
111	07	= read & write & execute

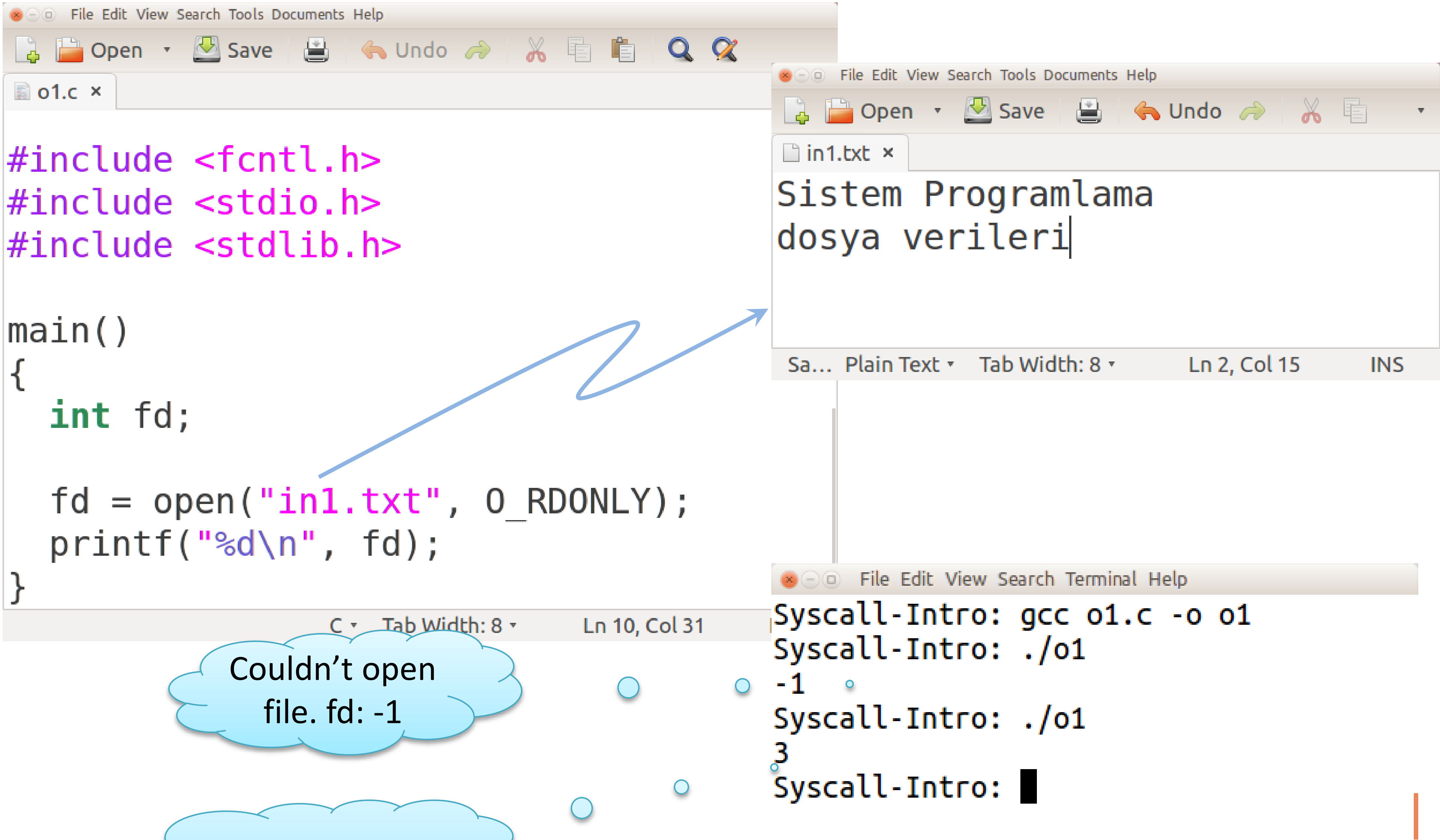
Example – open()

- Example: o1.c opens the file in1.txt for reading, and prints the value of the file descriptor.
- If you haven't copied over the file in1.txt, then it will print -1, since in1.txt does not exist.
- If in1.txt does exist, then it will print 3, meaning that the open() request has been granted (i.e. a non-negative integer was returned).



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

main()
{
  int fd;

  fd = open("in1.txt", O_RDONLY);
  printf("%d\n", fd);
}
```

Example – open()



o1.c:

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

main()
{
  int fd;

  fd = open("in1.txt", O_RDONLY);
  printf("%d\n", fd);
}
```

in1.txt:

```
Sistem Programlama
dosya verileri
```

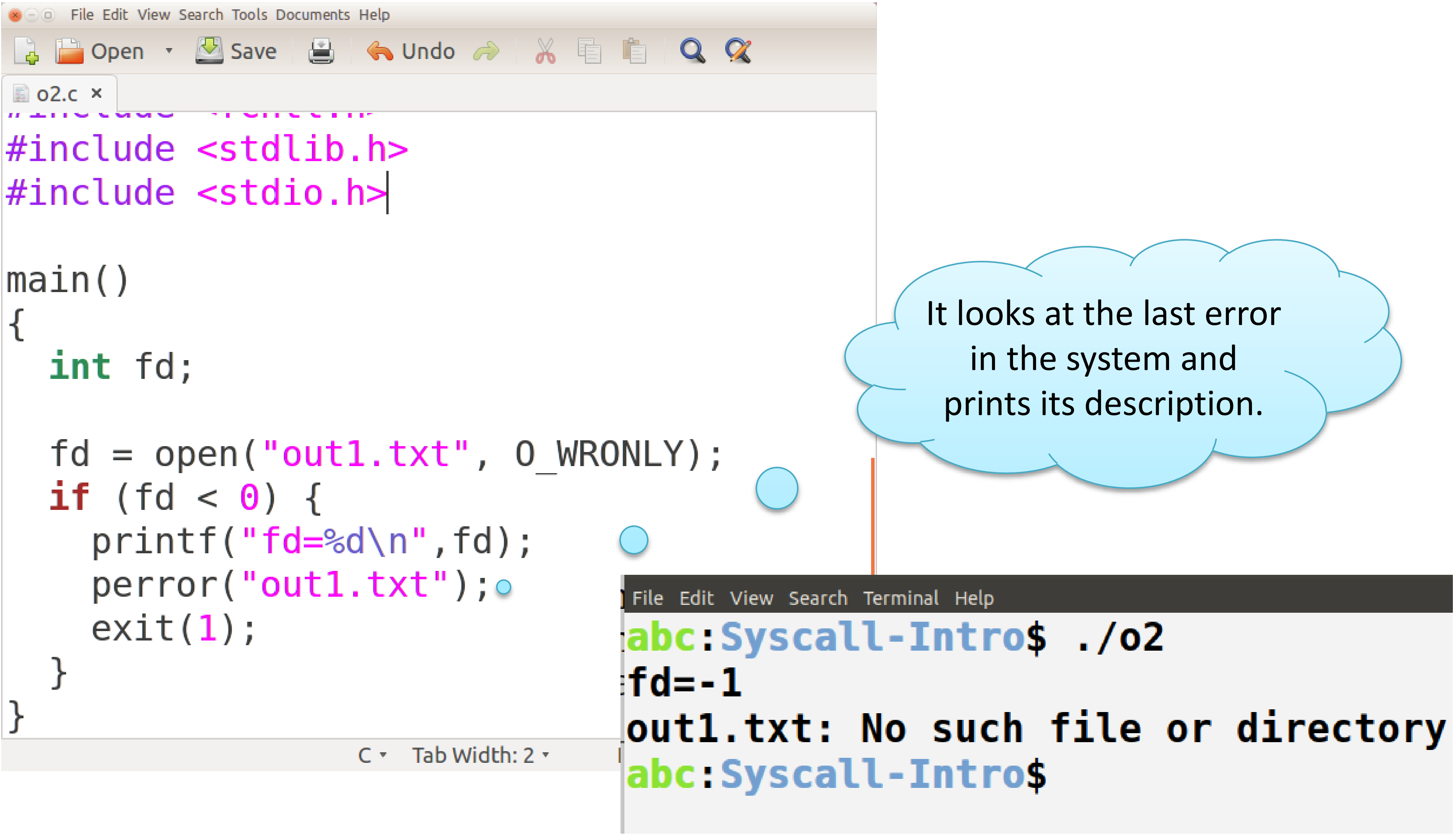
Terminal:

```
Syscall-Intro: gcc o1.c -o o1
Syscall-Intro: ./o1
-1
Syscall-Intro: ./o1
3
Syscall-Intro: 
```

Couldn’t open file. fd: -1

File opened. fd: 3

Example – open()

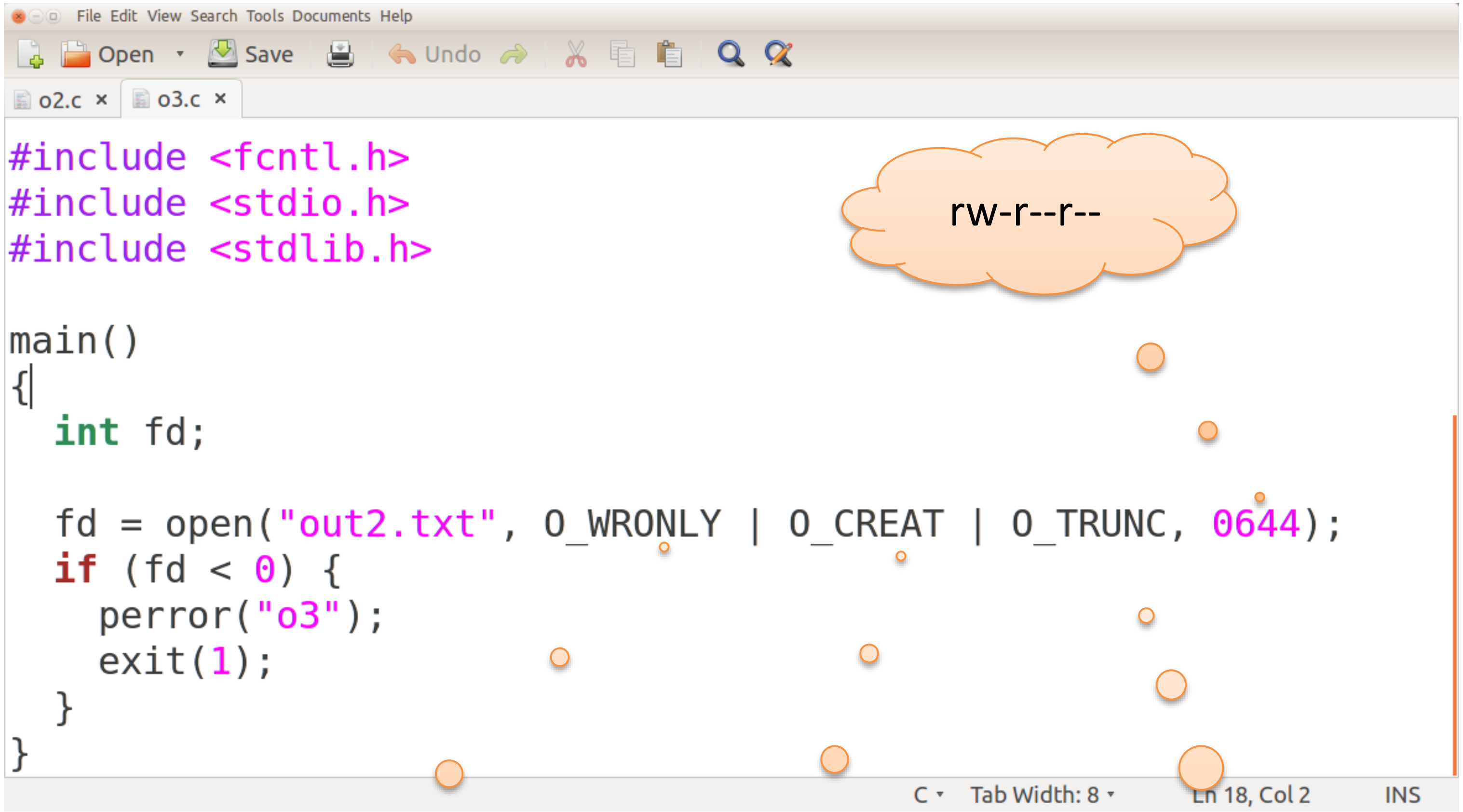

File Edit View Search Tools Documents Help
Open
Save
Undo
o2.c
C
Tab Width: 2
File Edit View Search Terminal Help

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

main()
{
  int fd;

  fd = open("out1.txt", O_WRONLY);
  if (fd < 0) {
    printf("fd=%d\n",fd);
    perror("out1.txt");
    exit(1);
  }
}
```

```
abc:Syscall-Intro$ ./o2
fd=-1
out1.txt: No such file or directory
abc:Syscall-Intro$
```

It looks at the last error in the system and prints its description.

Example – open()



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

main()
{
  int fd;

  fd = open("out2.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd < 0) {
    perror("o3");
    exit(1);
  }
}
```

rw-r--r--

Write only

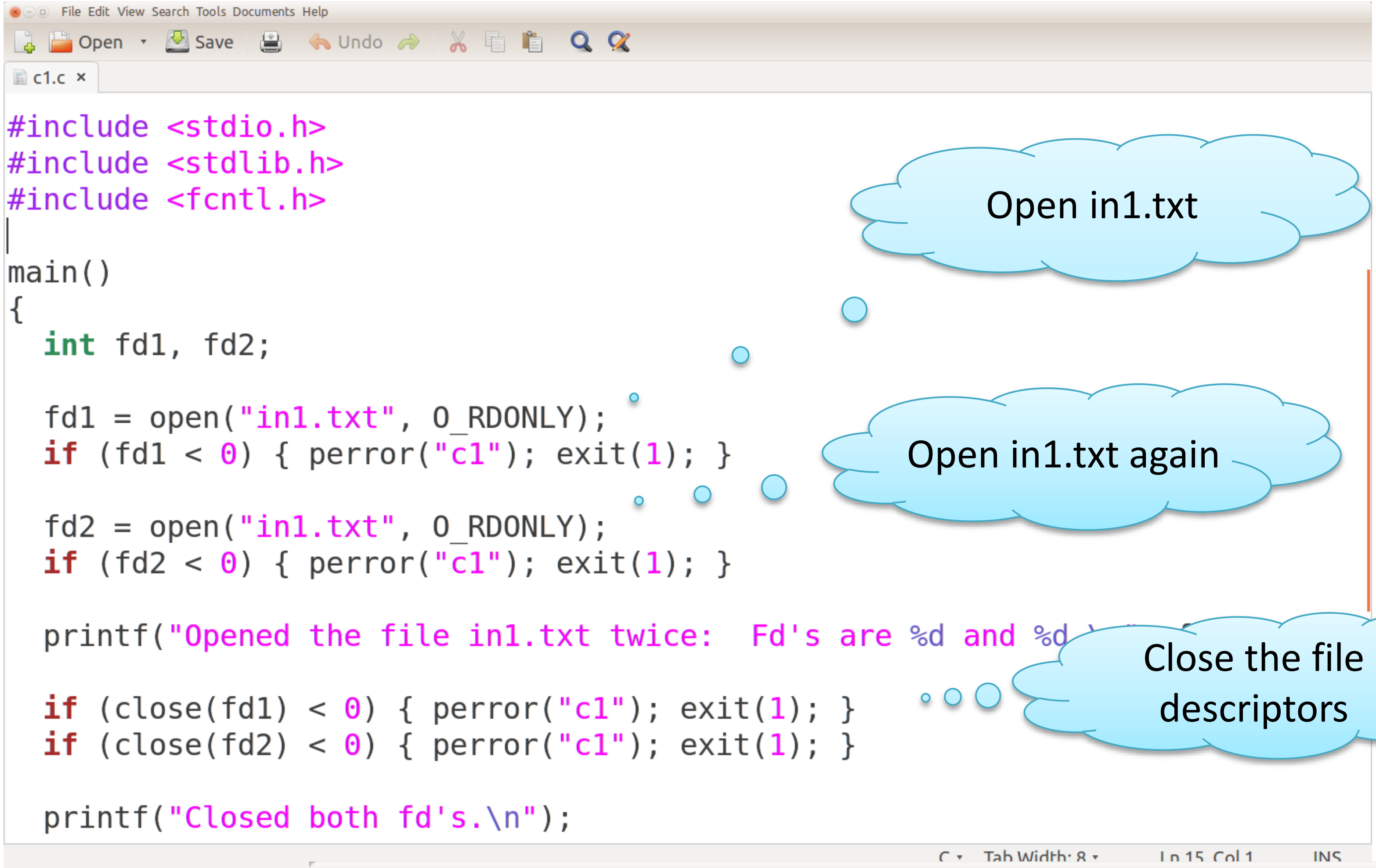
If file doesn't exist create a new file

If file exists, truncate the content.

Close

- **Close()** tells the operating system that you are done with a file descriptor. The OS can then reuse that file descriptor.
- The file c1.c shows some examples with opening and closing the file in1.txt. You should look at it carefully, as it opens the file multiple times without closing it, which is perfectly legal in Unix.

Example – close()

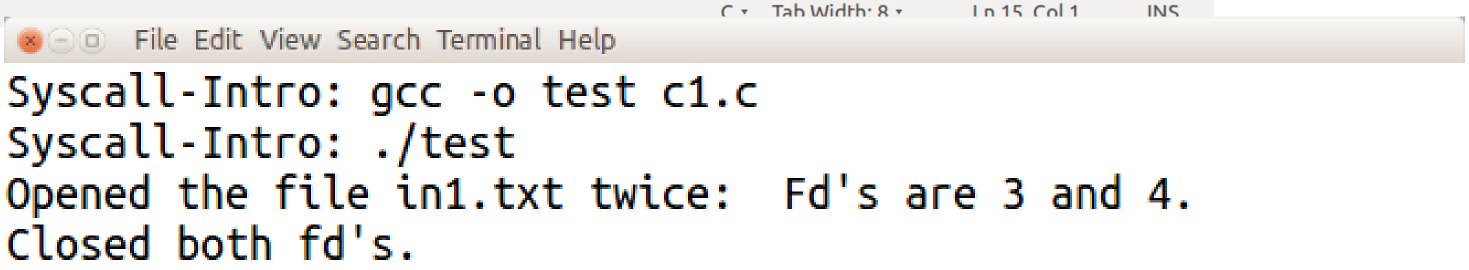

File Edit View Search Tools Documents Help
Open
Save
Undo
c1.c
C Tab Width: 8 Ln 15, Col 1 INS

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

main()
{
  int fd1, fd2;

  fd1 = open("in1.txt", O_RDONLY);
  if (fd1 < 0) { perror("c1"); exit(1); }

  fd2 = open("in1.txt", O_RDONLY);
  if (fd2 < 0) { perror("c1"); exit(1); }

  printf("Opened the file in1.txt twice:  Fd's are %d and %d \

  if (close(fd1) < 0) { perror("c1"); exit(1); }
  if (close(fd2) < 0) { perror("c1"); exit(1); }

  printf("Closed both fd's.\n");
```

Open in1.txt

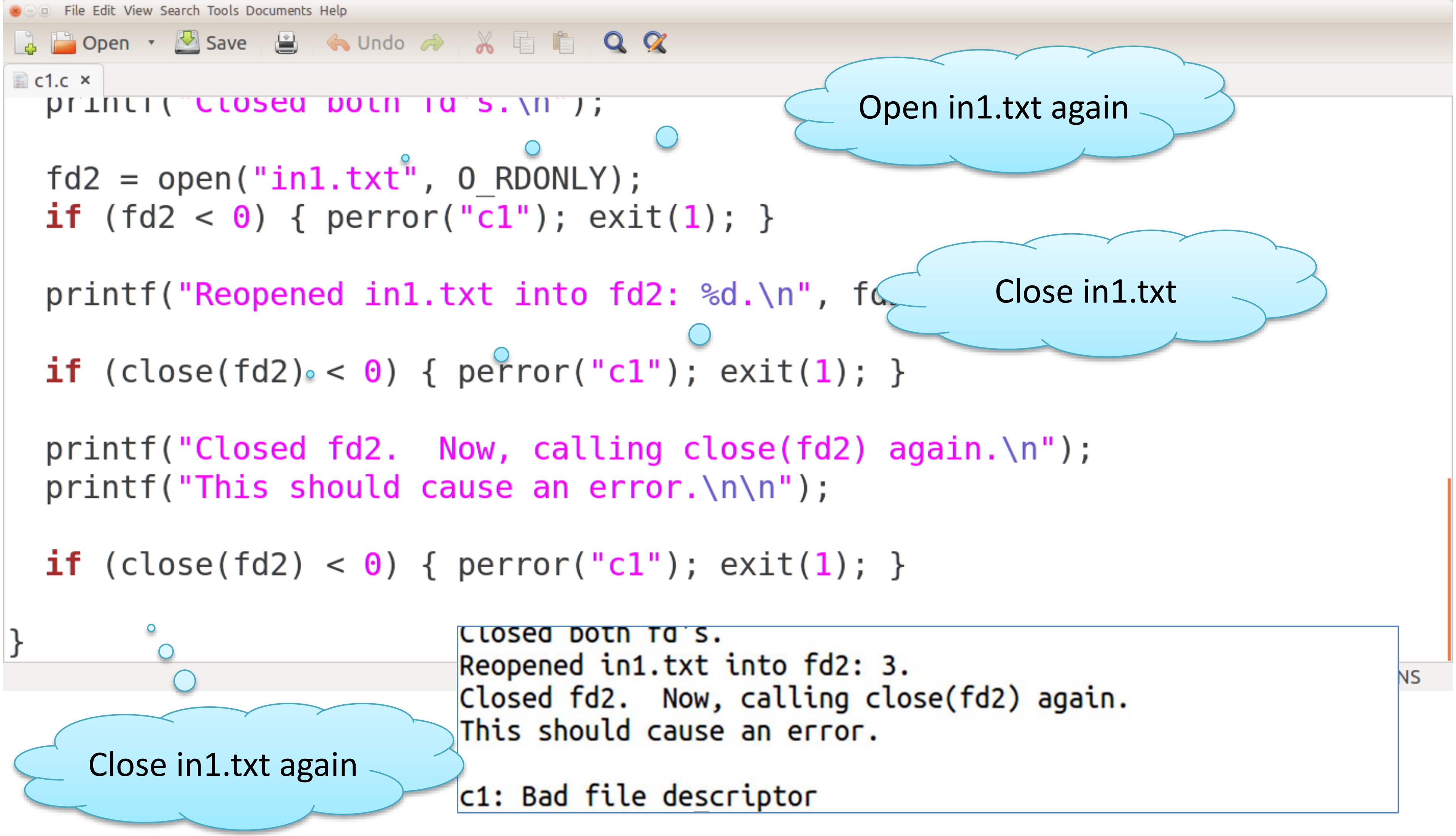
Open in1.txt again

Close the file descriptors



```
Syscall-Intro: gcc -o test c1.c
Syscall-Intro: ./test
Opened the file in1.txt twice:  Fd's are 3 and 4.
Closed both fd's.
```

Example – close()



```
  printf("Closed both fd's.\n");

  fd2 = open("in1.txt", O_RDONLY);
  if (fd2 < 0) { perror("c1"); exit(1); }

  printf("Reopened in1.txt into fd2: %d.\n", fd

  if (close(fd2) < 0) { perror("c1"); exit(1); }

  printf("Closed fd2.  Now, calling close(fd2) again.\n");
  printf("This should cause an error.\n\n");

  if (close(fd2) < 0) { perror("c1"); exit(1); }

}
```

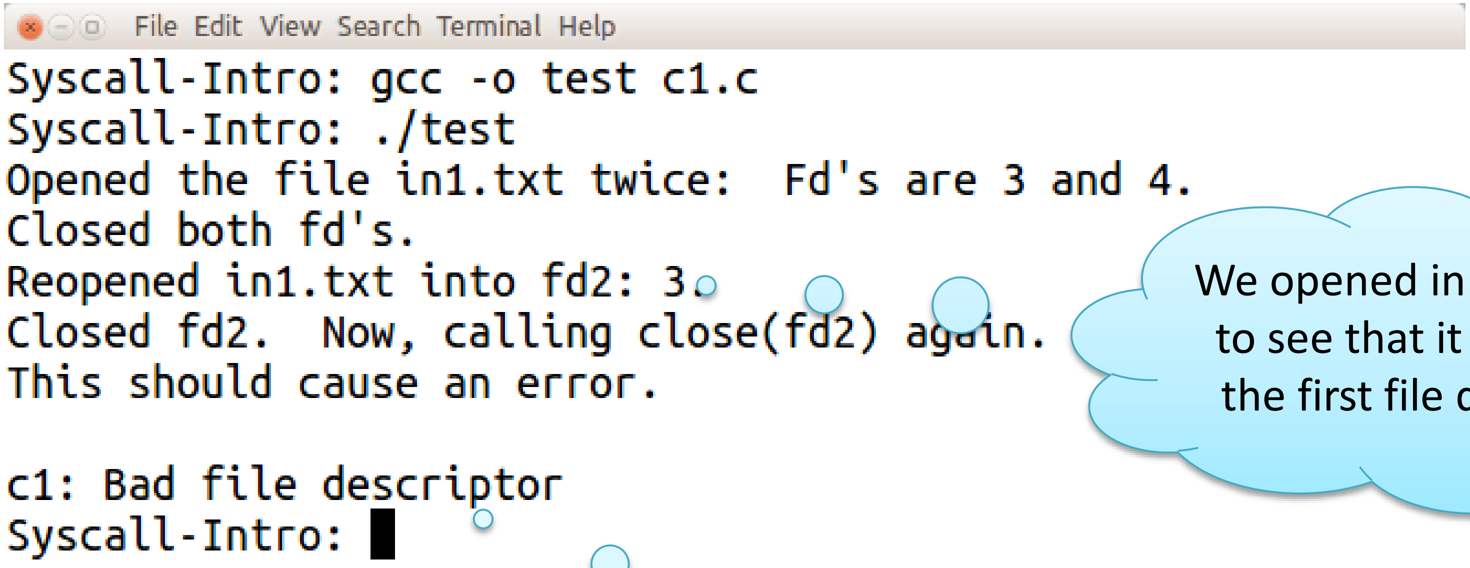
Open in1.txt again

Close in1.txt

Close in1.txt again

```
Closed both fd's.
Reopened in1.txt into fd2: 3.
Closed fd2.  Now, calling close(fd2) again.
This should cause an error.

c1: Bad file descriptor
```

Example – close()



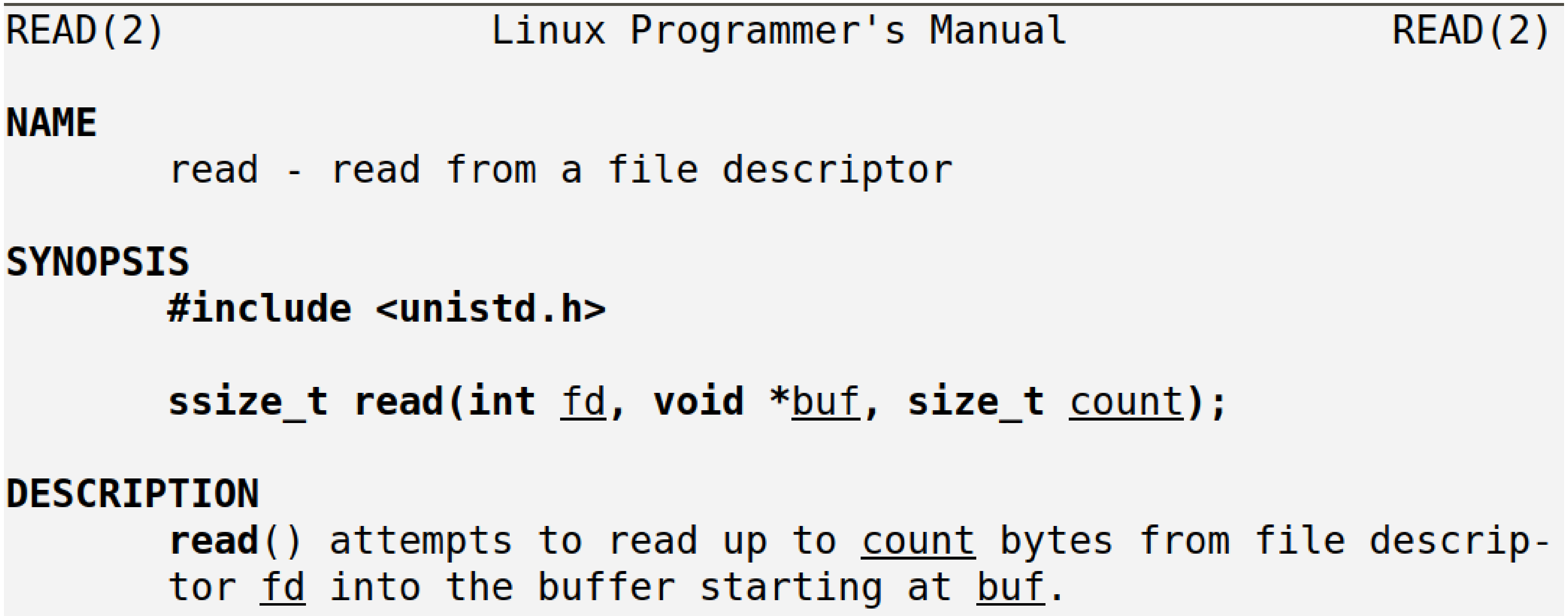
```
Syscall-Intro: gcc -o test c1.c
Syscall-Intro: ./test
Opened the file in1.txt twice:  Fd's are 3 and 4.
Closed both fd's.
Reopened in1.txt into fd2: 3.
Closed fd2.  Now, calling close(fd2) again.
This should cause an error.

c1: Bad file descriptor
Syscall-Intro: █
```

We opened in1.txt again, to see that it will reuse the first file descriptor

Close the file descriptor twice. The second causes an error.

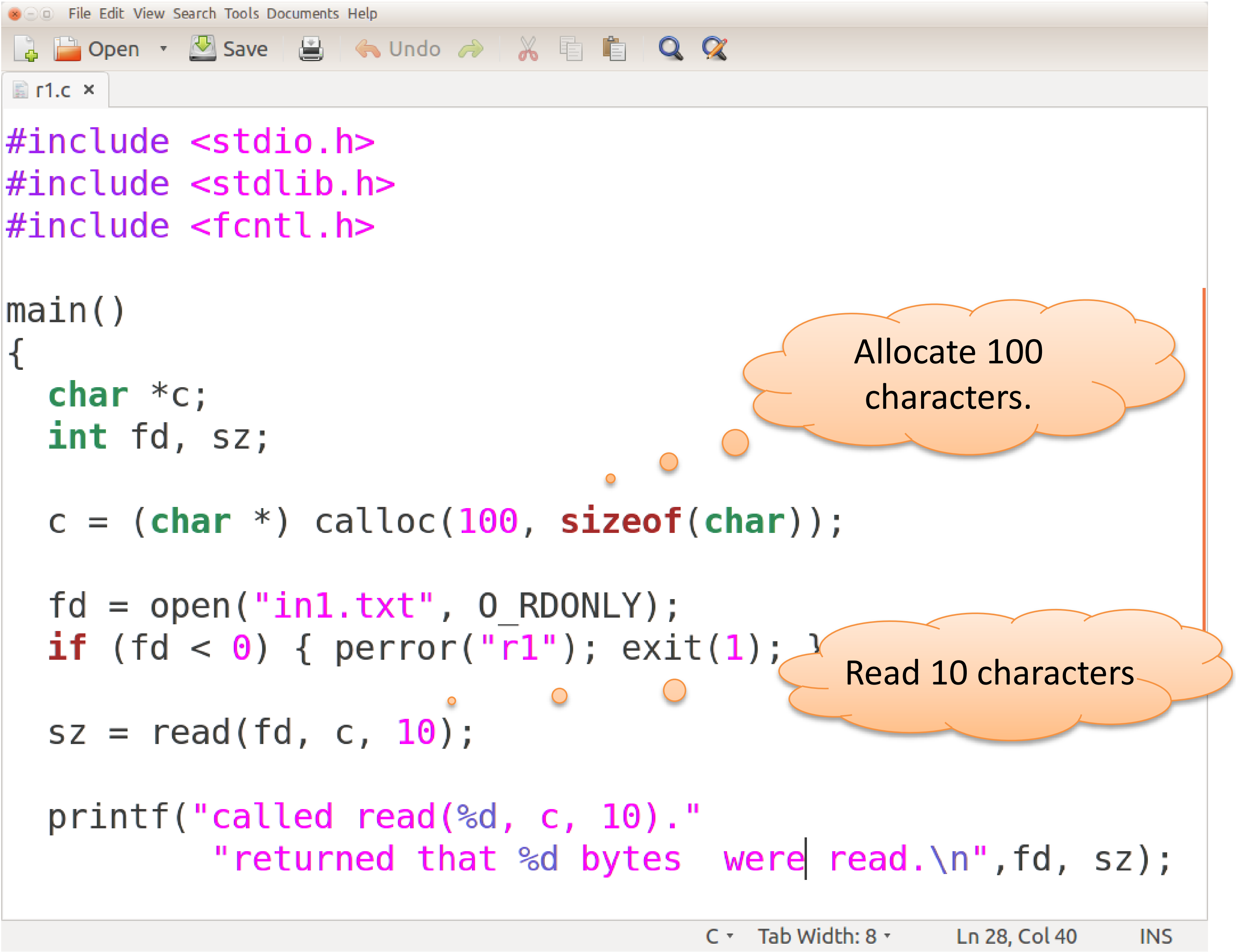
Read

- **Read()** tells the operating system to read "**count**" bytes from the file opened in file descriptor "**fd**", and to put those bytes into the location pointed to by "**buf**".
- It returns how many bytes were actually read.



```
READ(2)                 Linux Programmer's Manual                READ(2)

NAME
       read - read from a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t read(int fd, void *buf, size_t count);

DESCRIPTION
       read() attempts to read up to count bytes from file descrip-
       tor fd into the buffer starting at buf.
```

Example – read()



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

main()
{
  char *c;
  int fd, sz;

  c = (char *) calloc(100, sizeof(char));

  fd = open("in1.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);

  printf("called read(%d, c, 10)."
          "returned that %d bytes  were read.\n",fd, sz);
```

C   Tab Width: 8   Ln 28, Col 40   INS

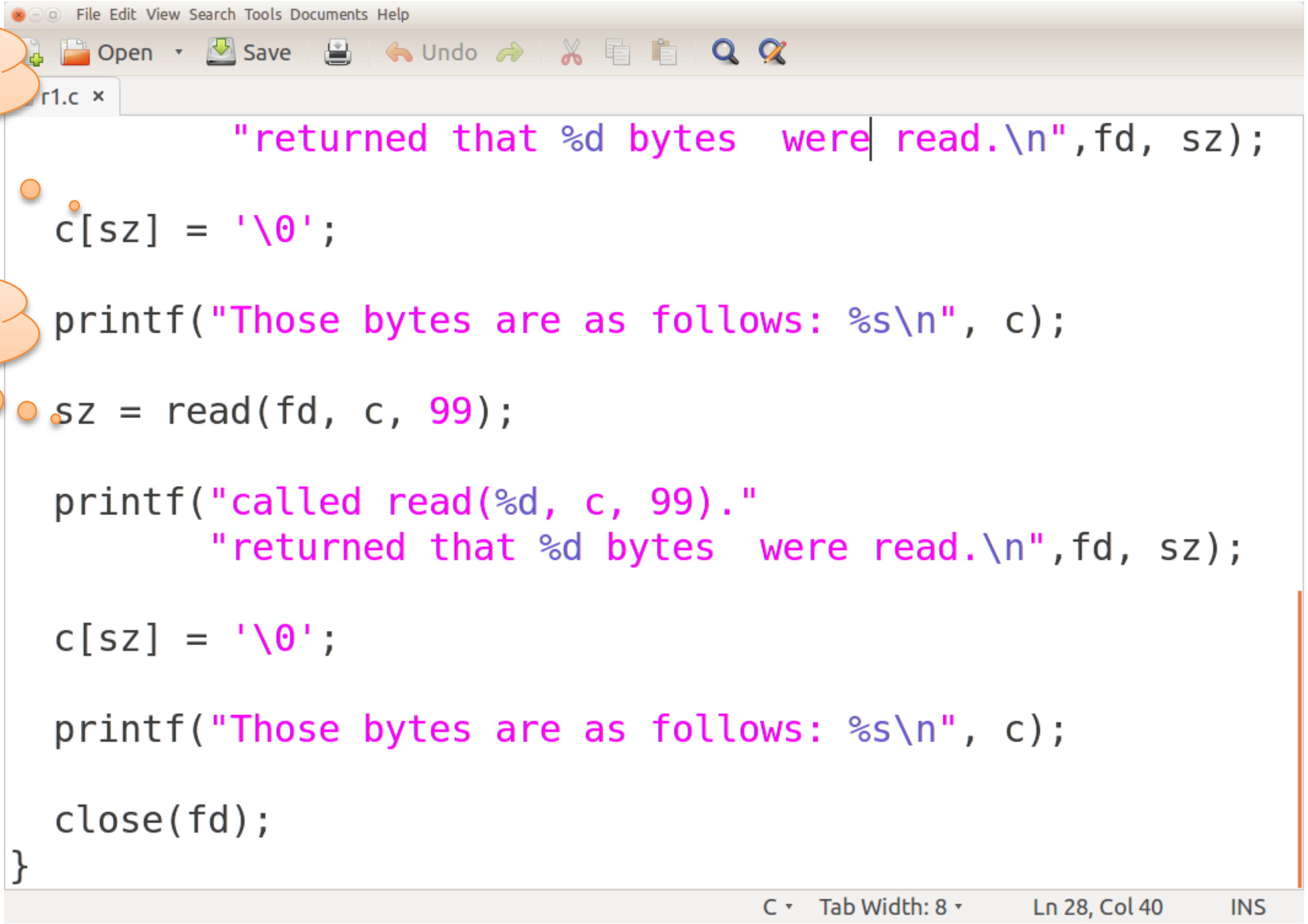
Allocate 100 characters.

Read 10 characters

Example – read()

Null termination

Read 99 characters.

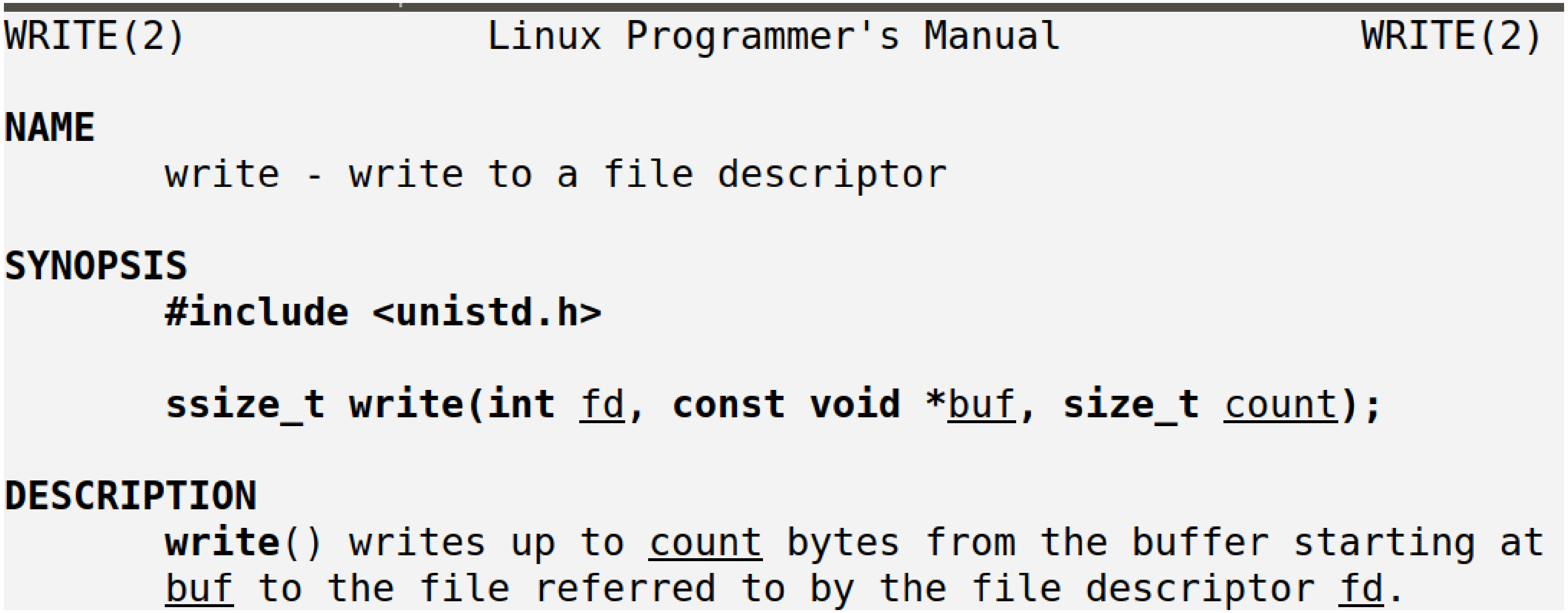


```
          "returned that %d bytes  were read.\n",fd, sz);

  c[sz] = '\0';

  printf("Those bytes are as follows: %s\n", c);

  sz = read(fd, c, 99);

  printf("called read(%d, c, 99)."
         "returned that %d bytes  were read.\n",fd, sz);

  c[sz] = '\0';

  printf("Those bytes are as follows: %s\n", c);

  close(fd);
}
```

C ▾ Tab Width: 8 ▾ Ln 28, Col 40 INS

write()

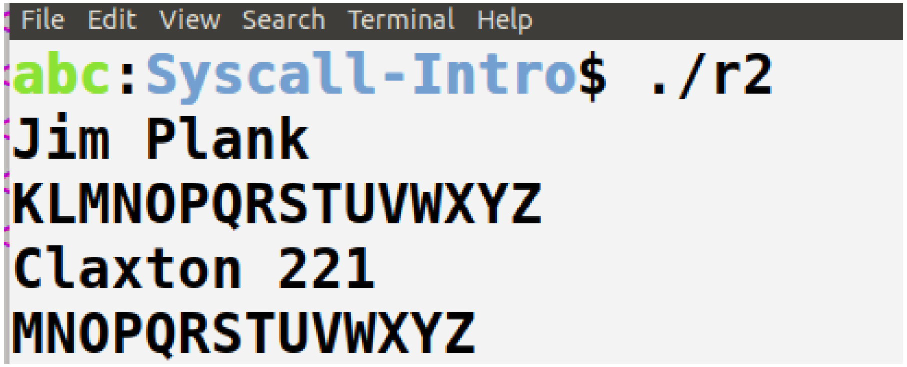
- Write() is just like read(), only it writes the bytes instead of reading them.
- It returns the number of bytes actually written, which is almost invariably "size".



```
WRITE(2)                  Linux Programmer's Manual                 WRITE(2)

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

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

DESCRIPTION
       write() writes up to count bytes from the buffer starting at
       buf to the file referred to by the file descriptor fd.
```

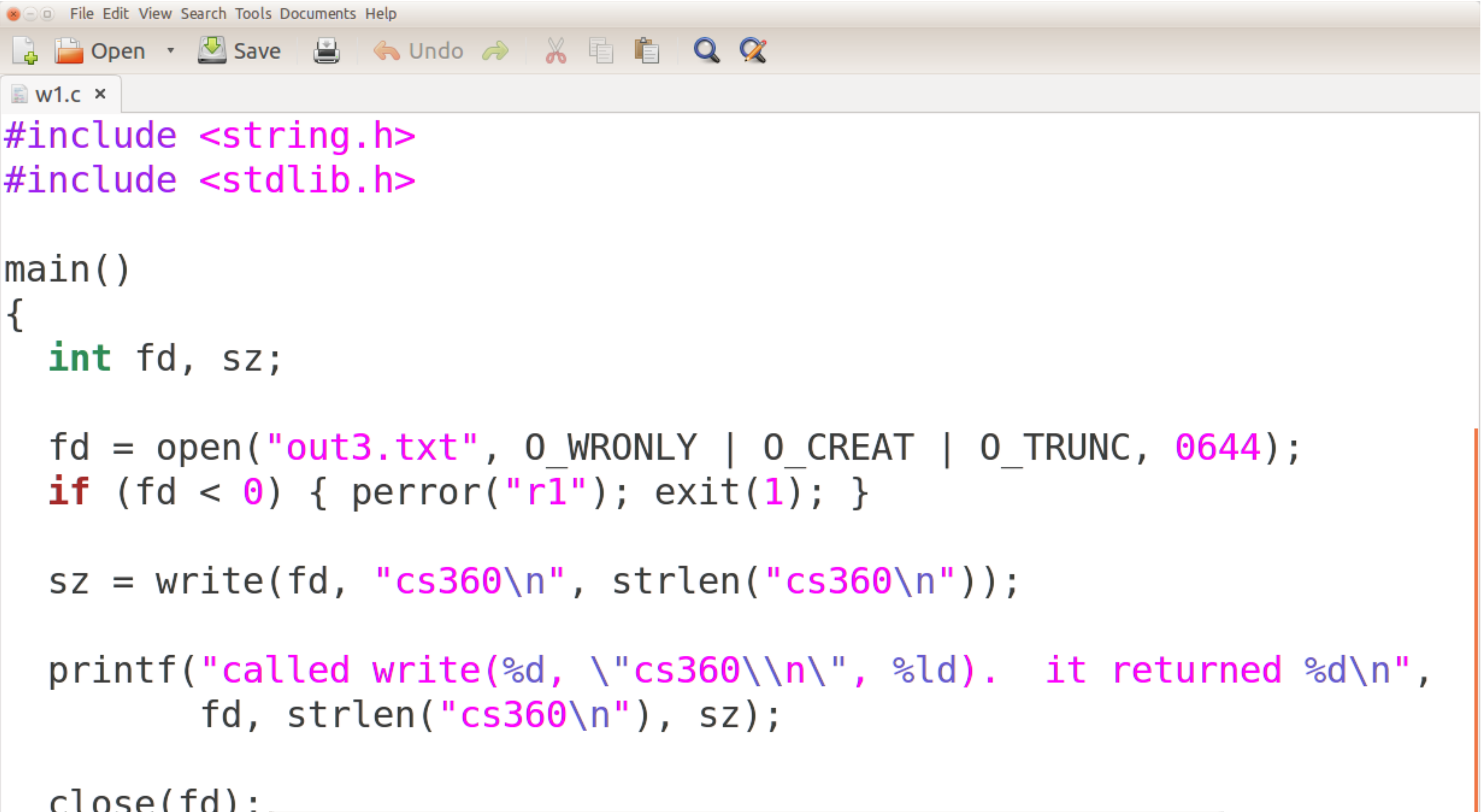
Example – write()

```c
/* Showing what happens when you don't NULL terminate. */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

int main()
{
  char c[100];
  int fd;

  strcpy(c, "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
  fd = open("in1.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  read(fd, c, 10);
  printf("%s\n", c);

  read(fd, c, 99);
  printf("%s\n", c);

  return 0;
```

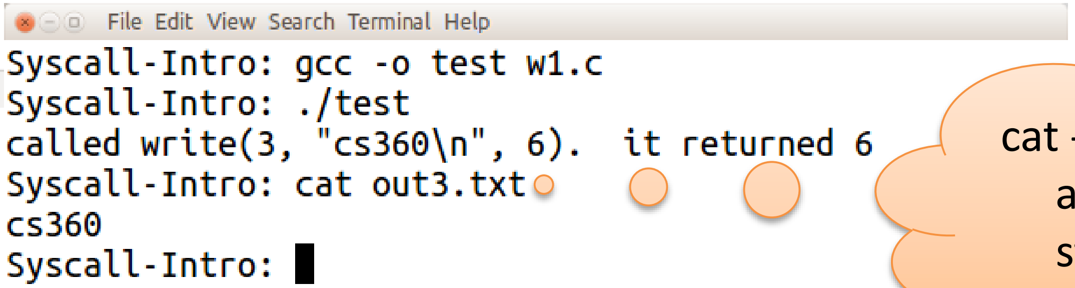


```
abc:Syscall-Intro$ ./r2
Jim Plank
KLMNOPQRSTUVWXYZ
Claxton 221
MNOPQRSTUVWXYZ
```

Write



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

main()
{
  int fd, sz;

  fd = open("out3.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = write(fd, "cs360\n", strlen("cs360\n"));

  printf("called write(%d, \"cs360\\n\", %ld).  it returned %d\n",
         fd, strlen("cs360\n"), sz);

  close(fd);
}
```

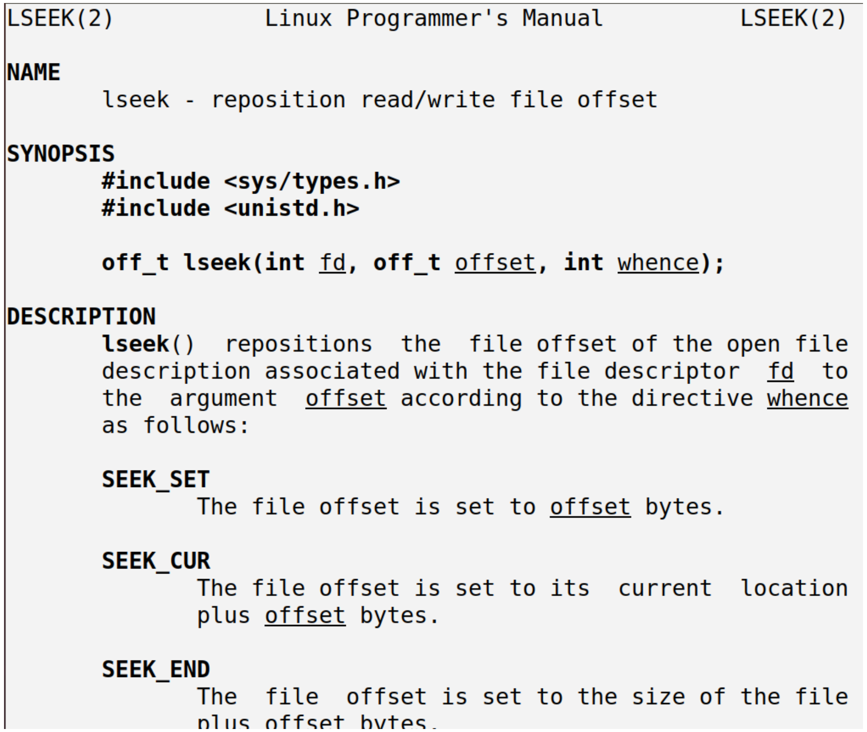


```
Syscall-Intro: gcc -o test w1.c
Syscall-Intro: ./test
called write(3, "cs360\n", 6).  it returned 6
Syscall-Intro: cat out3.txt
cs360
Syscall-Intro: 
```

cat - concatenate files and print on the standard output

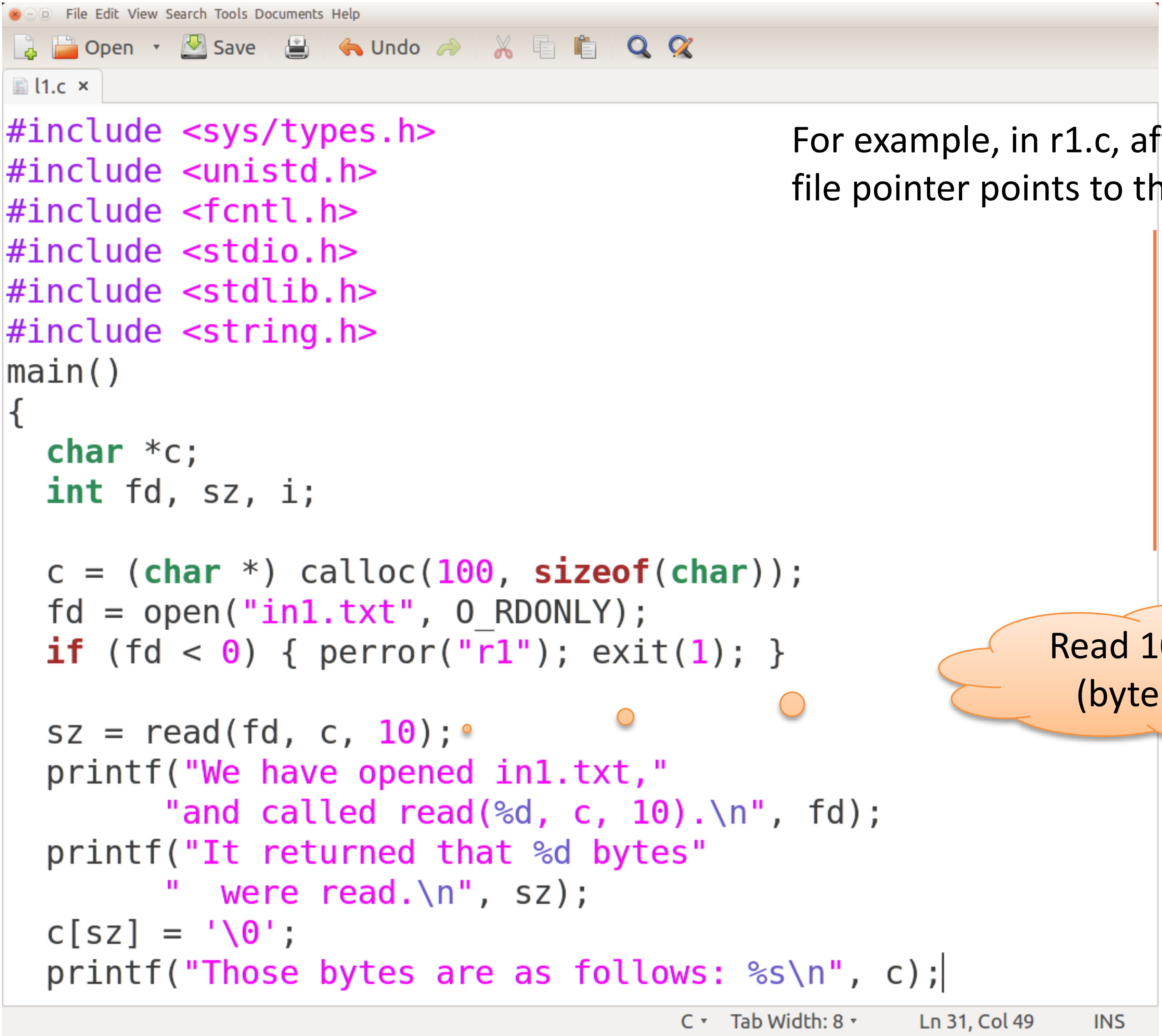
lseek()

- All open files have a "file pointer" associated with them. When the file is opened, the file pointer points to the beginning of the file.
- As the file is read or written, the file pointer moves.
- You can move the file pointer manually with lseek().
- The 'whence' variable of lseek specifies how the seek is to be done -- from the beginning of the file, from the current value of the pointer, and from the end of the file.
- The return value is the offset of the pointer after the lseek.



```
LSEEK(2)              Linux Programmer's Manual              LSEEK(2)

NAME
       lseek - reposition read/write file offset

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       off_t lseek(int fd, off_t offset, int whence);

DESCRIPTION
       lseek()  repositions  the  file offset of the open file
       description associated with the file descriptor  fd  to
       the  argument  offset according to the directive whence
       as follows:

       SEEK_SET
              The file offset is set to offset bytes.

       SEEK_CUR
              The file offset is set to its  current  location
              plus offset bytes.

       SEEK_END
              The  file  offset is set to the size of the file
              plus offset bytes.
```

Example – lseek()

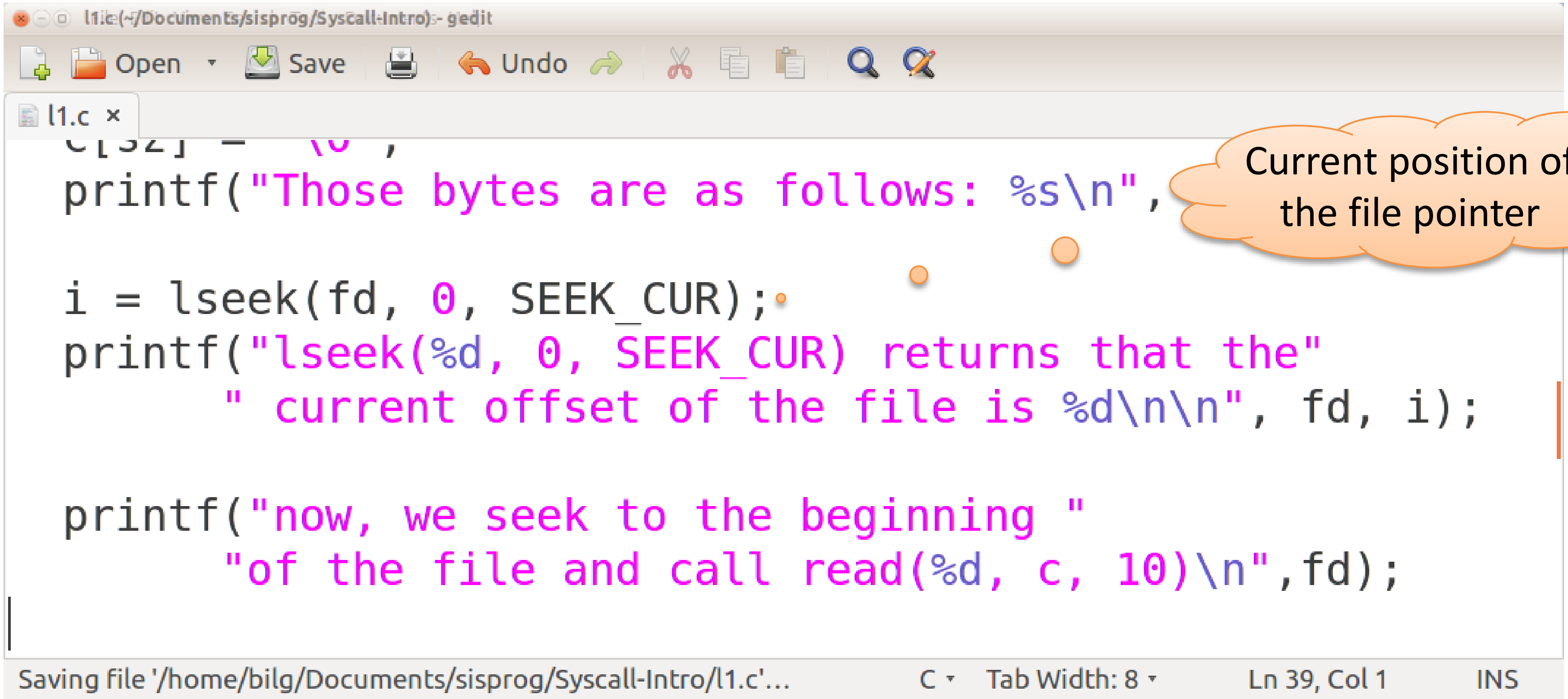


```c
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main()
{
  char *c;
  int fd, sz, i;

  c = (char *) calloc(100, sizeof(char));
  fd = open("in1.txt", O_RDONLY);
  if (fd < 0) { perror("r1"); exit(1); }

  sz = read(fd, c, 10);
  printf("We have opened in1.txt,"
        "and called read(%d, c, 10).\n", fd);
  printf("It returned that %d bytes"
        "  were read.\n", sz);
  c[sz] = '\0';
  printf("Those bytes are as follows: %s\n", c);
```

For example, in r1.c, after the first read, the file pointer points to the 11th byte in in1.txt.

Read 10 characters (byte) from file.

Example – lseek()

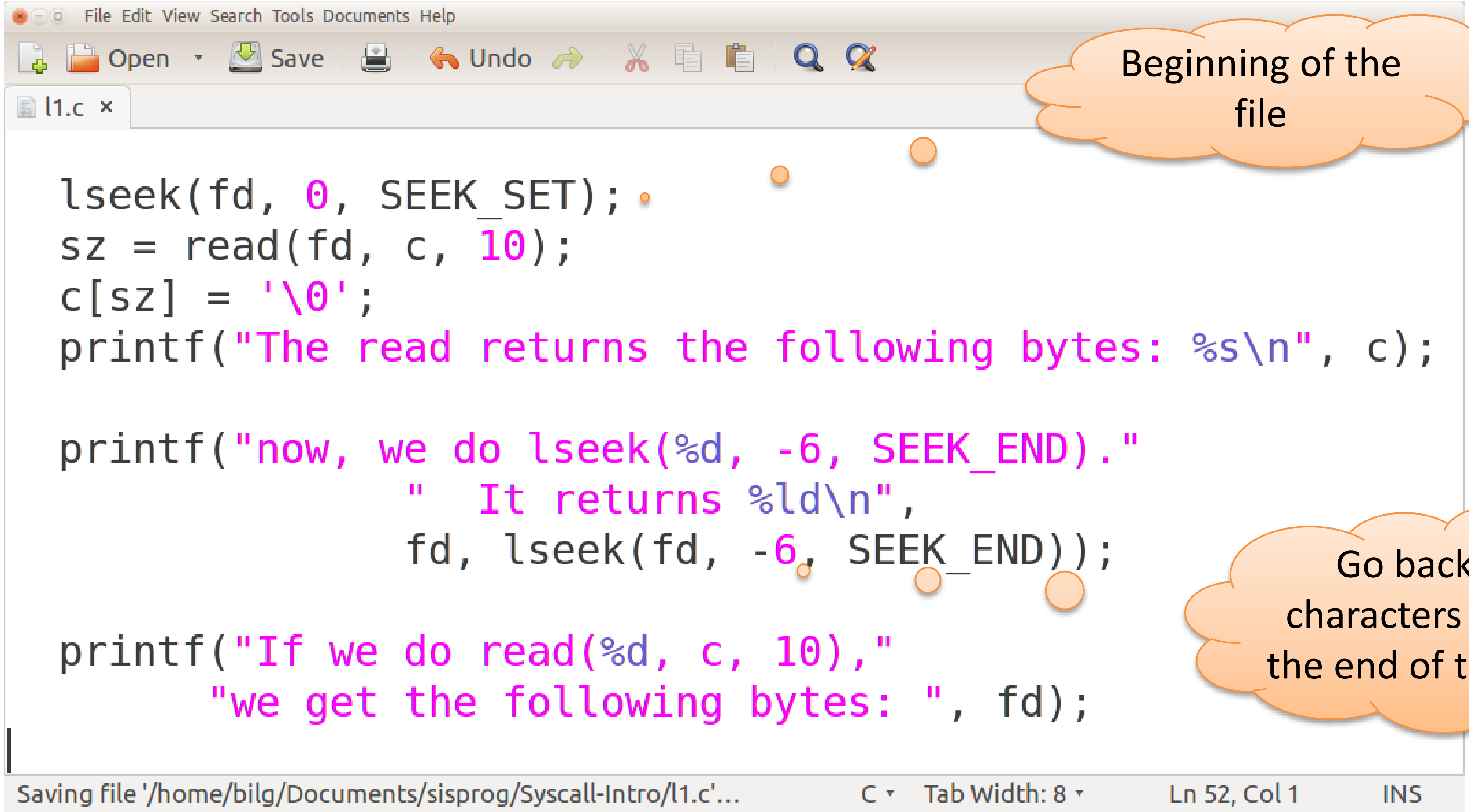


```
  c[32] = '\0';
  printf("Those bytes are as follows: %s\n",

  i = lseek(fd, 0, SEEK_CUR);
  printf("lseek(%d, 0, SEEK_CUR) returns that the"
         " current offset of the file is %d\n\n", fd, i);

  printf("now, we seek to the beginning "
         "of the file and call read(%d, c, 10)\n",fd);

```

Saving file '/home/bilg/Documents/sisprog/Syscall-Intro/l1.c'... C Tab Width: 8 Ln 39, Col 1 INS

Current position of the file pointer

Example – lseek()



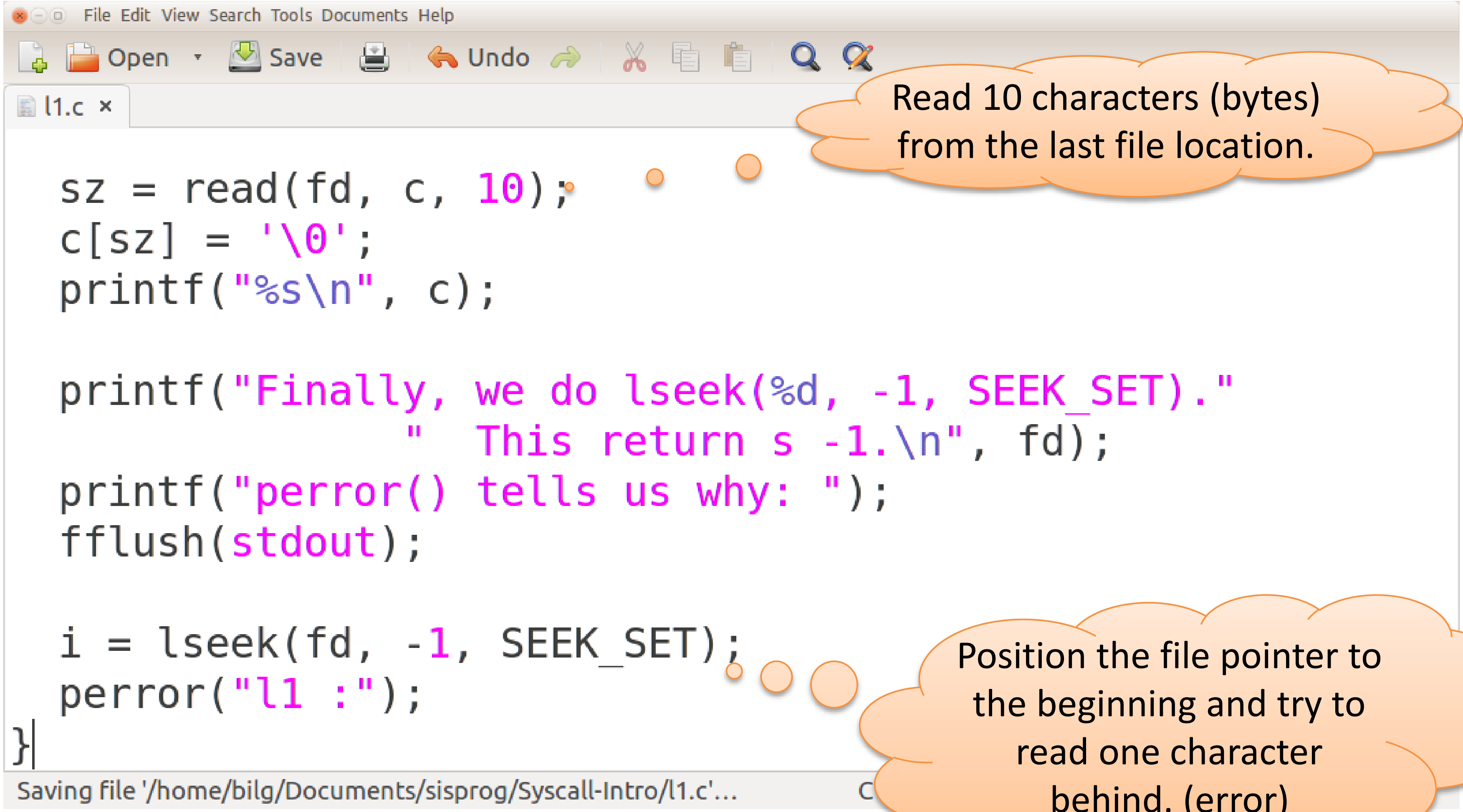
```c
lseek(fd, 0, SEEK_SET);
sz = read(fd, c, 10);
c[sz] = '\0';
printf("The read returns the following bytes: %s\n", c);

printf("now, we do lseek(%d, -6, SEEK_END)."
              "  It returns %ld\n",
              fd, lseek(fd, -6, SEEK_END));

printf("If we do read(%d, c, 10),"
      "we get the following bytes: ", fd);
```

Saving file '/home/bilg/Documents/sisprog/Syscall-Intro/l1.c'...

Beginning of the file

Go back 6 characters from the end of the file

Example – lseek()



```
  sz = read(fd, c, 10);
  c[sz] = '\0';
  printf("%s\n", c);

  printf("Finally, we do lseek(%d, -1, SEEK_SET)."
               "  This return s -1.\n", fd);
  printf("perror() tells us why: ");
  fflush(stdout);

  i = lseek(fd, -1, SEEK_SET);
  perror("l1 :");
}
```

Saving file '/home/bilg/Documents/sisprog/Syscall-Intro/l1.c'... C

Read 10 characters (bytes) from the last file location.

Position the file pointer to the beginning and try to read one character behind. (error)

Standard Input, Standard Output, and Standard Error

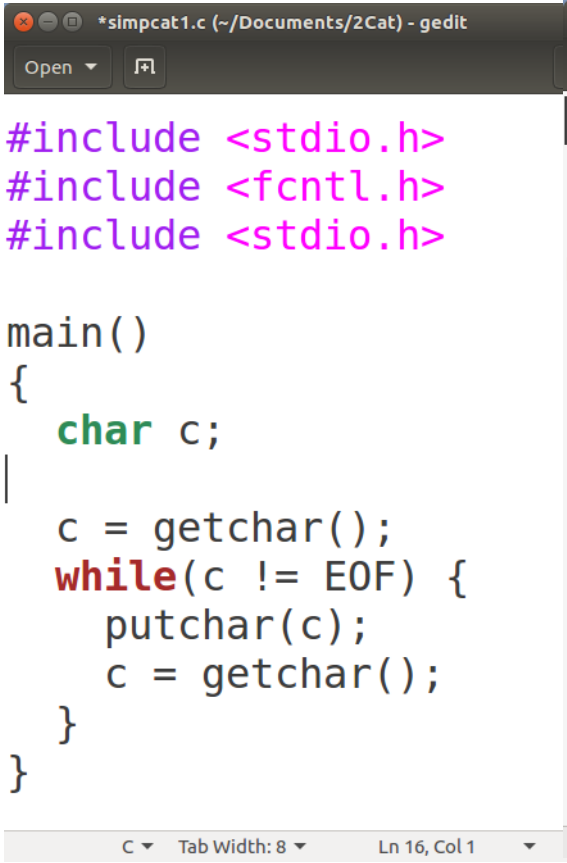
- Now, every process in Unix starts out with three file descriptors predefined and open:

- File descriptor 0 is standard input.
- File descriptor 1 is standard output.
- File descriptor 2 is standard error.

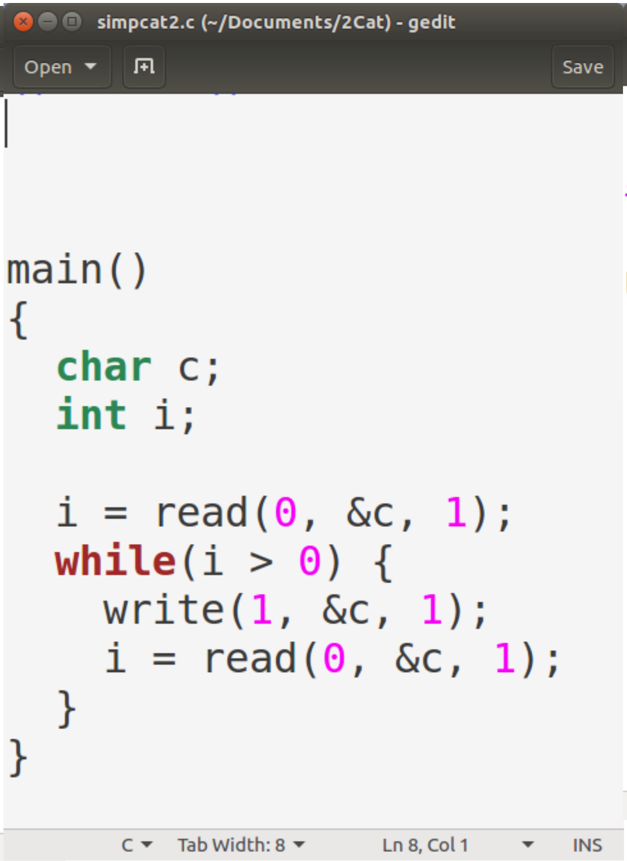
- Thus, when you write a program, you can read from standard input, using read(0, ...), and write to standard output using write(1, ...).

Simpcat

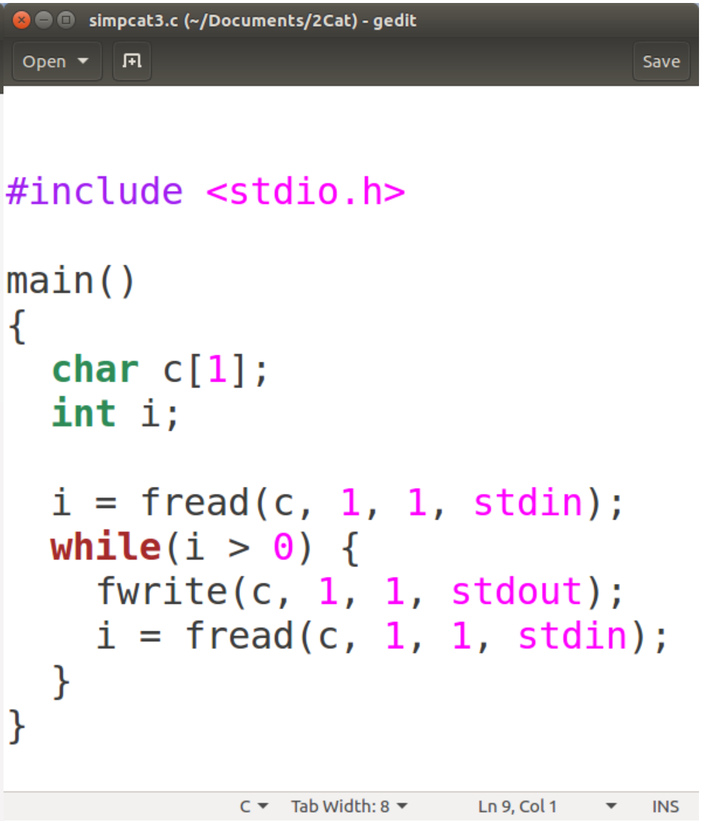
- we can write a very simple cat program (one that copies standard input to standard output) with one line.
- Here are three equivalent ways of writing a simple cat program, which just reads from standard input, and writes to standard output.



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

main()
{
   char c;

   c = getchar();
   while(c != EOF) {
      putchar(c);
      c = getchar();
   }
}
```



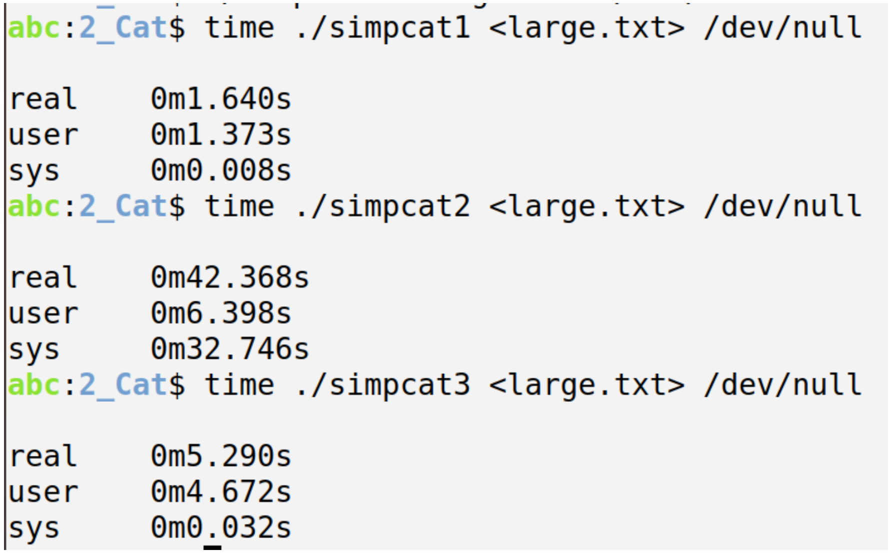
```c
main()
{
   char c;
   int i;

   i = read(0, &c, 1);
   while(i > 0) {
      write(1, &c, 1);
      i = read(0, &c, 1);
   }
}
```



```c
#include <stdio.h>

main()
{
   char c[1];
   int i;

   i = fread(c, 1, 1, stdin);
   while(i > 0) {
      fwrite(c, 1, 1, stdout);
      i = fread(c, 1, 1, stdin);
   }
}
```

Simpcat

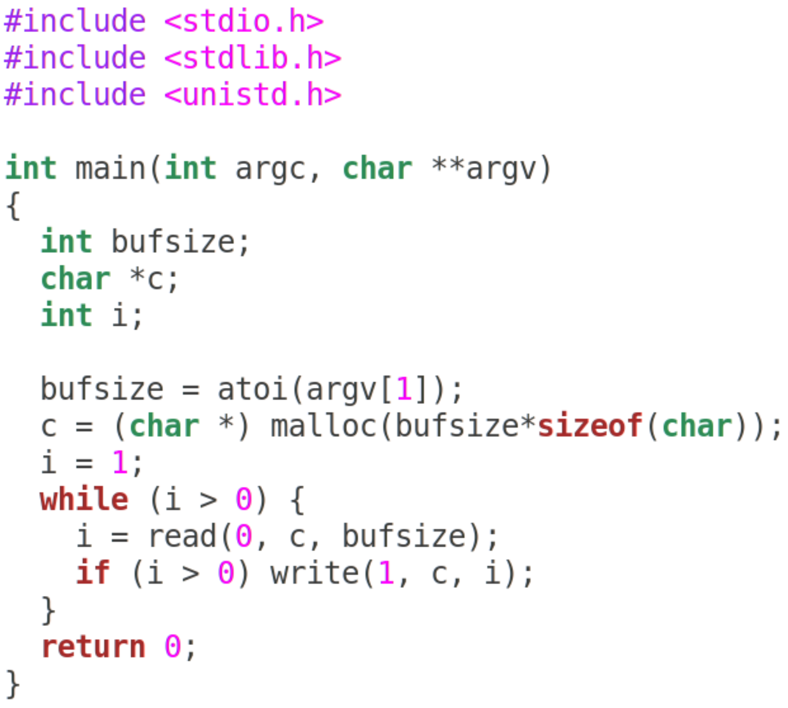
- So, what's going on? /dev/null is a special file in Unix that you can write to, but it never stores anything on disk.
- We're using it so that you don't create 25M files in your home directory as this wastes disk space. "Large.txt" is a 25,000,000-byte file. This means that in simpcat1.c, getchar() and putchar() are being called 25 million times each, as are read() and write() in simpcat2.c, and fread() and fwrite() in simpcat3.c.
- Obviously, the culprit in simpcat2.c is the fact that the program is making system calls instead of library calls. Remember that a system call is a request made to the operating system. This means at each read/write call, the operating system has to take over the CPU (this means saving the state of the simpcat2 program), process the request, and return (which means restoring the state of the simpcat2 program).
- This is evidently far more expensive than what simpcat1.c and simpcat3.c do.



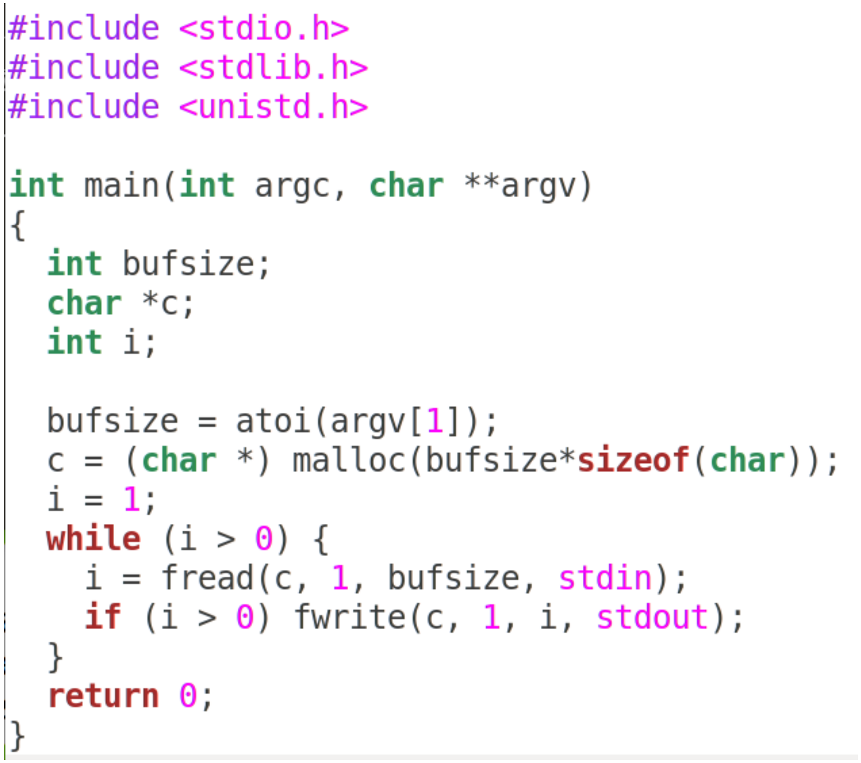
```
abc:2_Cat$ time ./simpcat1 <large.txt> /dev/null

real	0m1.640s
user	0m1.373s
sys	0m0.008s
abc:2_Cat$ time ./simpcat2 <large.txt> /dev/null

real	0m42.368s
user	0m6.398s
sys	0m32.746s
abc:2_Cat$ time ./simpcat3 <large.txt> /dev/null

real	0m5.290s
user	0m4.672s
sys	0m0.032s
```

Simpcat

- This program copies standard input to standard output using read/write   and buffering.
- The buffer size is specified by the user.



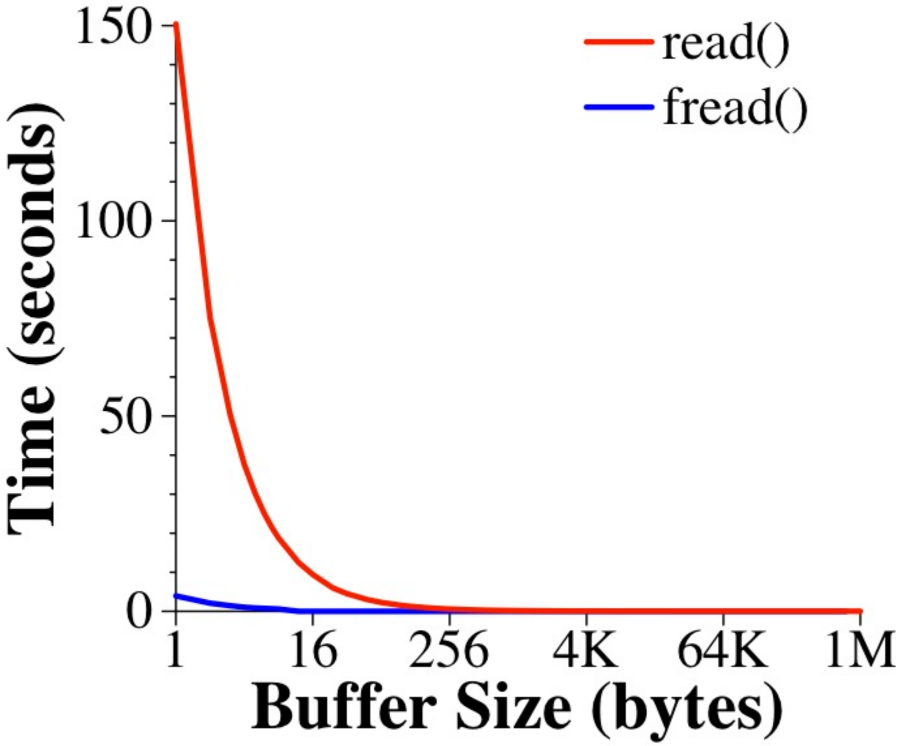
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int bufsize;
  char *c;
  int i;

  bufsize = atoi(argv[1]);
  c = (char *) malloc(bufsize*sizeof(char));
  i = 1;
  while (i > 0) {
    i = read(0, c, bufsize);
    if (i > 0) write(1, c, i);
  }
  return 0;
}
```



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

int main(int argc, char **argv)
{
  int bufsize;
  char *c;
  int i;

  bufsize = atoi(argv[1]);
  c = (char *) malloc(bufsize*sizeof(char));
  i = 1;
  while (i > 0) {
    i = fread(c, 1, bufsize, stdin);
    if (i > 0) fwrite(c, 1, i, stdout);
  }
  return 0;
}
```

Simpcat

- These let us read in more than one byte at a time.
- This is called buffering: You allocate a region of memory in which to store things, so that you can make fewer system/procedure calls.
- Note that fread() and fwrite() are just like read() and write(), except that they go to the standard I/O library instead of the operating s



Simpcat

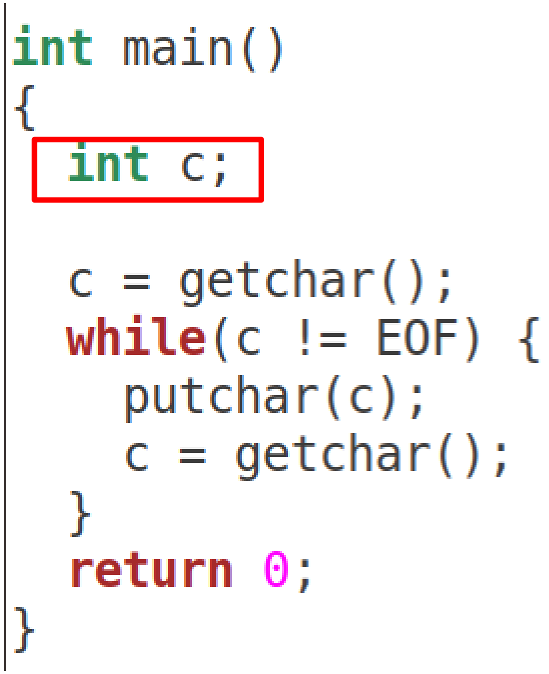
- **What's the lesson behind this?**
- 1. Buffering is a good way to cut down on too many system calls.
- 2. If you are reading small chunks of bytes, then use **getchar()** or **fread()**. They do buffering for you.
- 3. If you are doing single character I/O, use **getchar()** (or **fgetc()**).
- 4. If you are reading large chunks of bytes, then **fread()** and **read()** work about the same. However, you should use **fread()**, since it makes your programming more consistent, and because it does a little more error checking for you.
- The same is true for writes, even though we didn't go through them in detail in class.

Standard I/O vs System calls

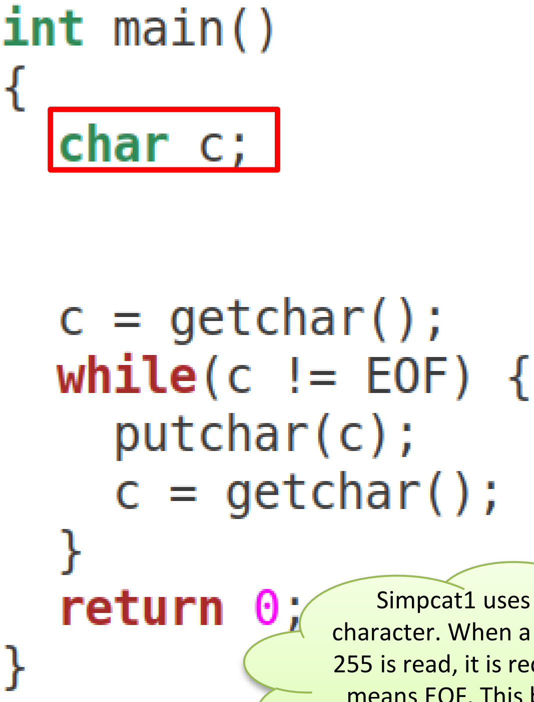
Each system call has analogous procedure calls from the standard I/O library:

System Call	Standard I/O call
open	fopen
close	fclose
read/write	getchar/putchar
	getc/putc
	fgetc/fputc
	fread/fwrite
	gets/puts
	fgets/fputs
	scanf/printf
	fscanf/fprintf
lseek	fseek

Chars vs ints

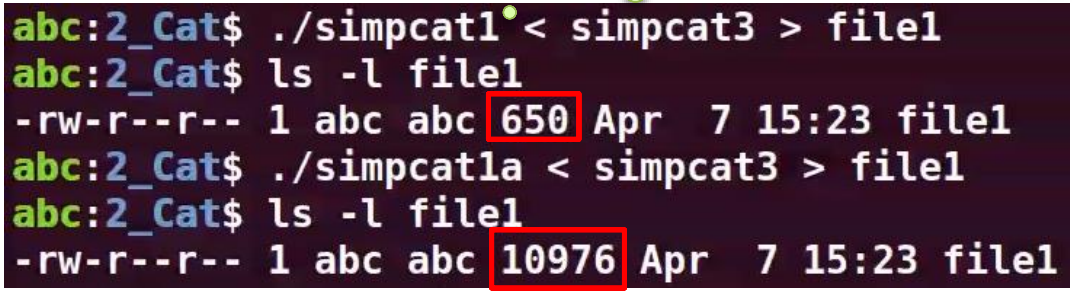


```
int main()
{
  int c;

  c = getchar();
  while(c != EOF) {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```



```
int main()
{
  char c;


  c = getchar();
  while(c != EOF) {
    putchar(c);
    c = getchar();
  }
  return 0;
}
```

Simpcat1 uses a char to copy character. When a byte that contains 255 is read, it is recorded as -1 which means EOF. This breaks while loop and copying stops.



```
abc:2_Cat$ ./simpcat1 < simpcat3 > file1
abc:2_Cat$ ls -l file1
-rw-r--r-- 1 abc abc 650 Apr  7 15:23 file1
abc:2_Cat$ ./simpcat1a < simpcat3 > file1
abc:2_Cat$ ls -l file1
-rw-r--r-- 1 abc abc 10976 Apr  7 15:23 file1
```