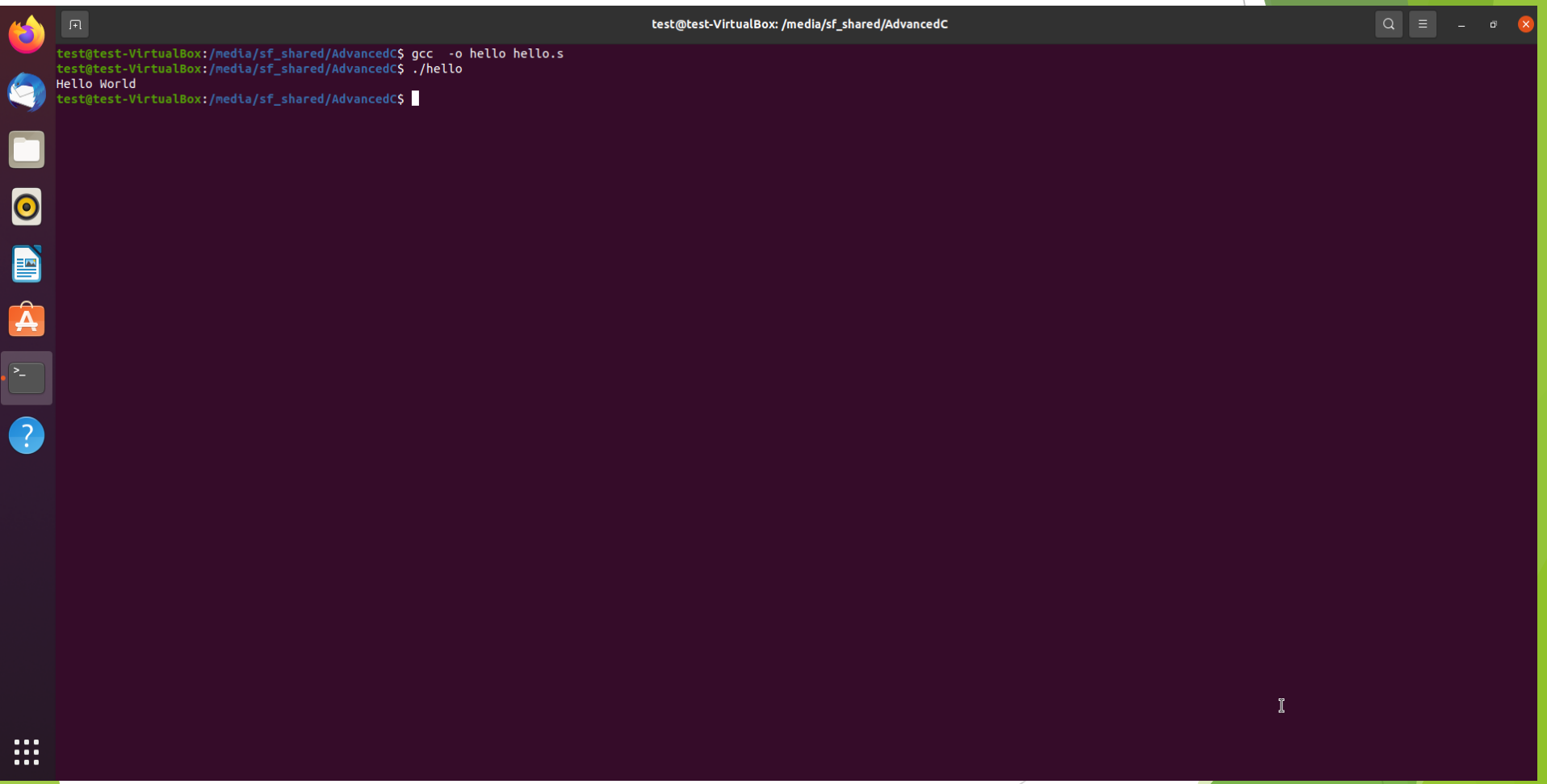
Advanced C and System Programming

Anandkumar

X86 Assembly

```
# ----------------------------------------------------------------------------------------
# Writes "Hello World" to the console using a C library. Runs on Linux or any other system
# that does not use underscores for symbols in its C library. To assemble and run:
#
#     gcc -o hello hello.s && ./hello
# ----------------------------------------------------------------------------------------
        .global main

        .text
main:                           # This is called by C library's startup code
        push %rbp
       mov %rsp,%rbp
        lea           message(%rip), %rdi       # First integer (or pointer) parameter in %rdi
        call    printf              # printf(message)
        pop %rbp
        ret
                                # Return to C library code

                .data
message:
        .asciz "Hello World\n"          # asciz puts a 0 byte at the end
```

X86 Assembly



X86 Assembly

The 64-bit calling conventions are a bit more detailed, and they are explained fully in the AMD64 ABI Reference:

•From left to right, pass as many parameters as will fit in registers. The order in which registers are allocated, are:

- For integers and pointers, rdi, rsi, rdx, rcx, r8, r9.
- For floating-point (float, double), xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7

X86 Assembly

- Additional parameters are pushed on the stack, right to left, and are removed by the caller after the call.
- The only registers that the called function is required to preserve (the calle-save registers) are: rbp, rbx, r12, r13, r14, r15. All others are free to be changed by the called function.
- Integers are returned in rax or rdx:rax, and floating point values are returned in xmm0 or xmm1:xmm0.

X86 Assembly

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes

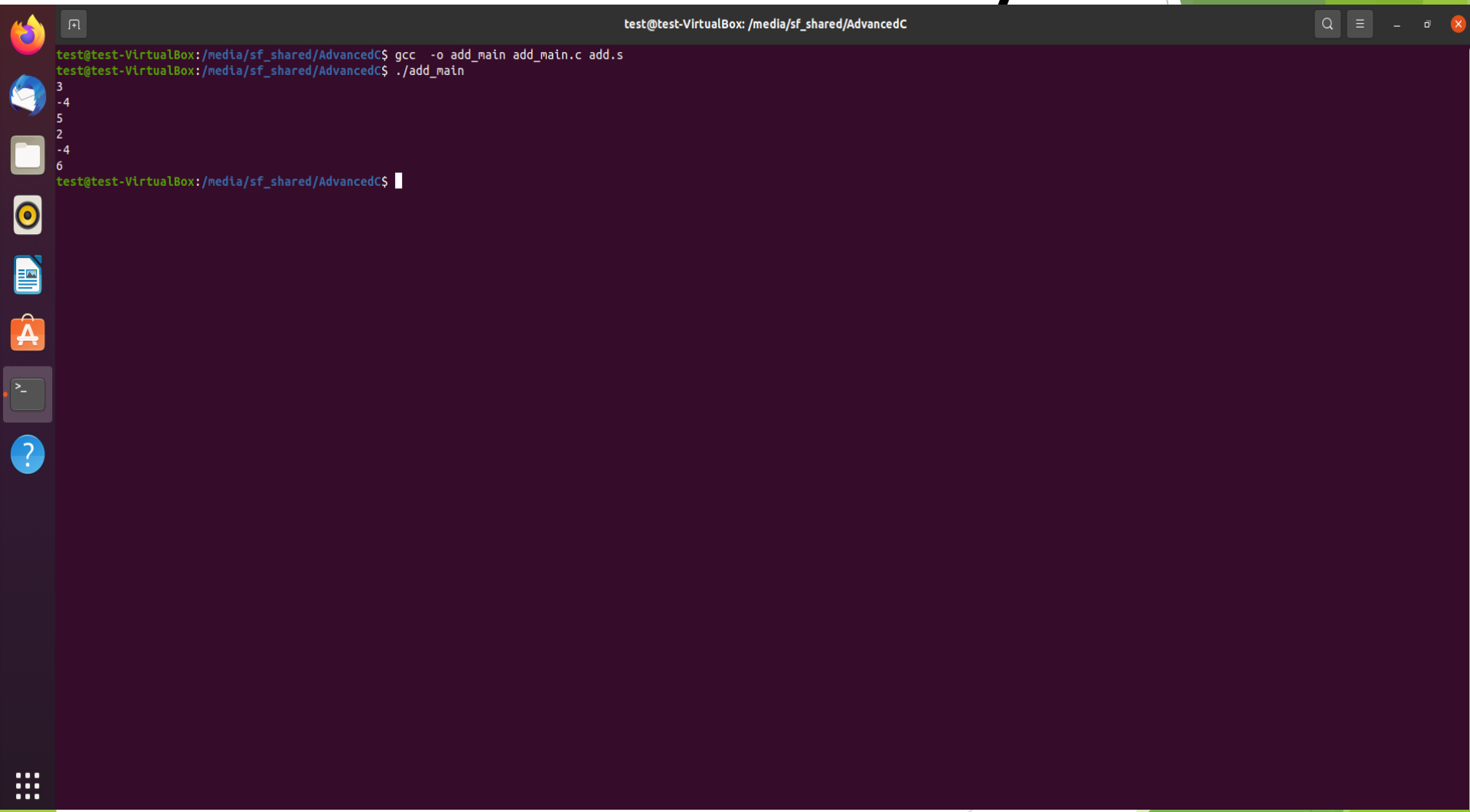
X86 Assembly

```
/*
 * add_main.c
 *
 * A small program that illustrates how to call the add function we wrote in
 * assembly language.
 */

#include <stdio.h>

int add(int, int);

int main() {
    printf("%d\n", add(1, 2));
    printf("%d\n", add(2, -6));
    printf("%d\n", add(2, 3));
    printf("%d\n", add(-2, 4));
    printf("%d\n", add(2, -6));
    printf("%d\n", add(2, 4));
    return 0;
}
```

X86 Assembly

```
# -----------------------------------------------------------------------------
# A 64-bit function that returns the addition of two number.
# The function has signature:
#
#   int add(int x, int y)
#
# Note that the parameters have already been passed in rdi and rsi.  We
# just have to return the value in rax.
# -----------------------------------------------------------------------------

        .globl  add

        .text
add:
        push %rbp
        mov %rsi,%rax
        add %rdi,%rax
        pop %rbp
         ret
```

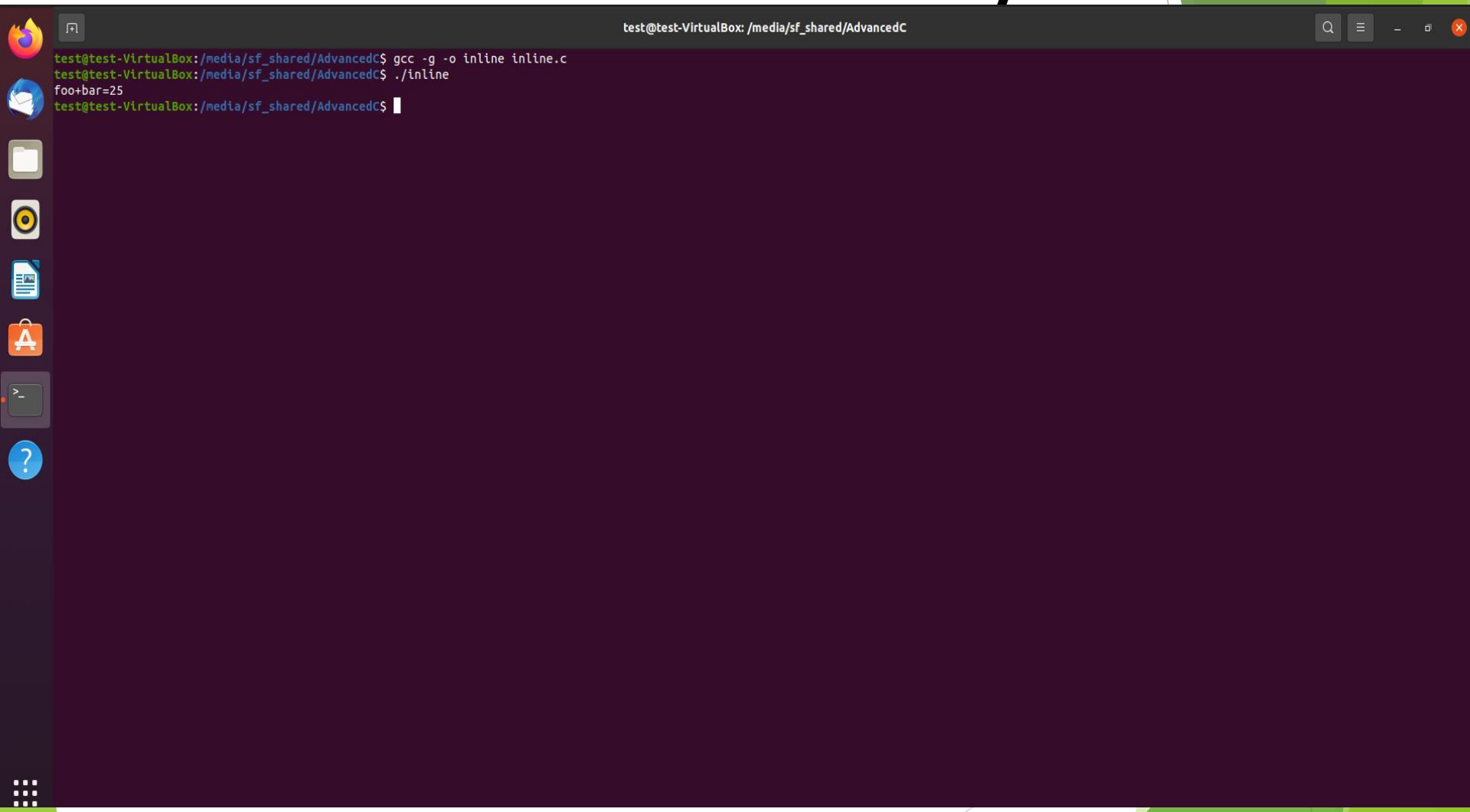
X86 Assembly



X86 Assembly

```
/*
 * inline.c
 *
 * A small program that illustrates how to inline assembly in C code
 */
#include <stdio.h>

int main(void)
{
	int foo = 10, bar = 15;
	asm("addl  %%ebx,%%eax"
			:"=a"(foo)
			:"a"(foo), "b"(bar)
			);
	printf("foo+bar=%d\n", foo);
	return 0;
}
```

X86 Assembly



Libraries in Linux

Libraries in Linux

- Virtually all programs in Linux are linked against one or more libraries.

- Libraries contain code and data that provide services to independent programs.

- There are two types of library in Linux:

    - Static libraries (Archive file, same as windows .LIB file).

    - Shared libraries (Shared Object, same as windows DLL).

Static Library

- Is a collection of object files.

- Linker extracts needed object files from archive and attaches them to your program (as they were provided directly).

- When linker encounters an archive in command line, it searches the already passed objects to see if there is a reference to objects in this archive or not.

Static Library

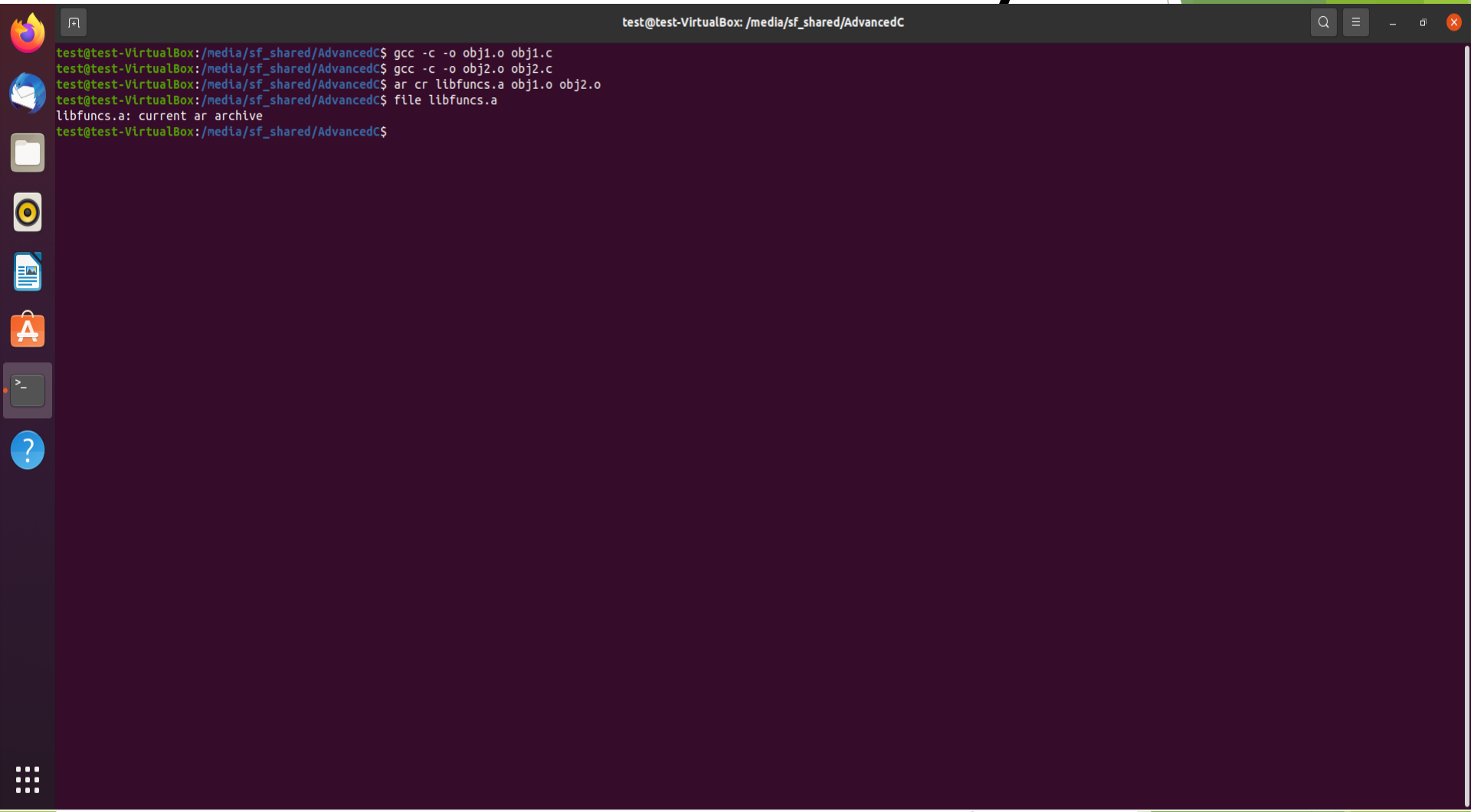
Obj1.c

```
--------
#include<stdio.h>

void my_func1()
{
printf("This is func1\n");
}
```

Obj2.c

```
---------
#include<stdio.h>

void my_func2()
{
printf("This is func2\n");
}
```

Static Library



Static Library

```
#include<stdio.h>
void my_func1(void);
void my_func2(void);

int main()
{
my_func1();
my_func2();
return 0;
}
```

Static Library

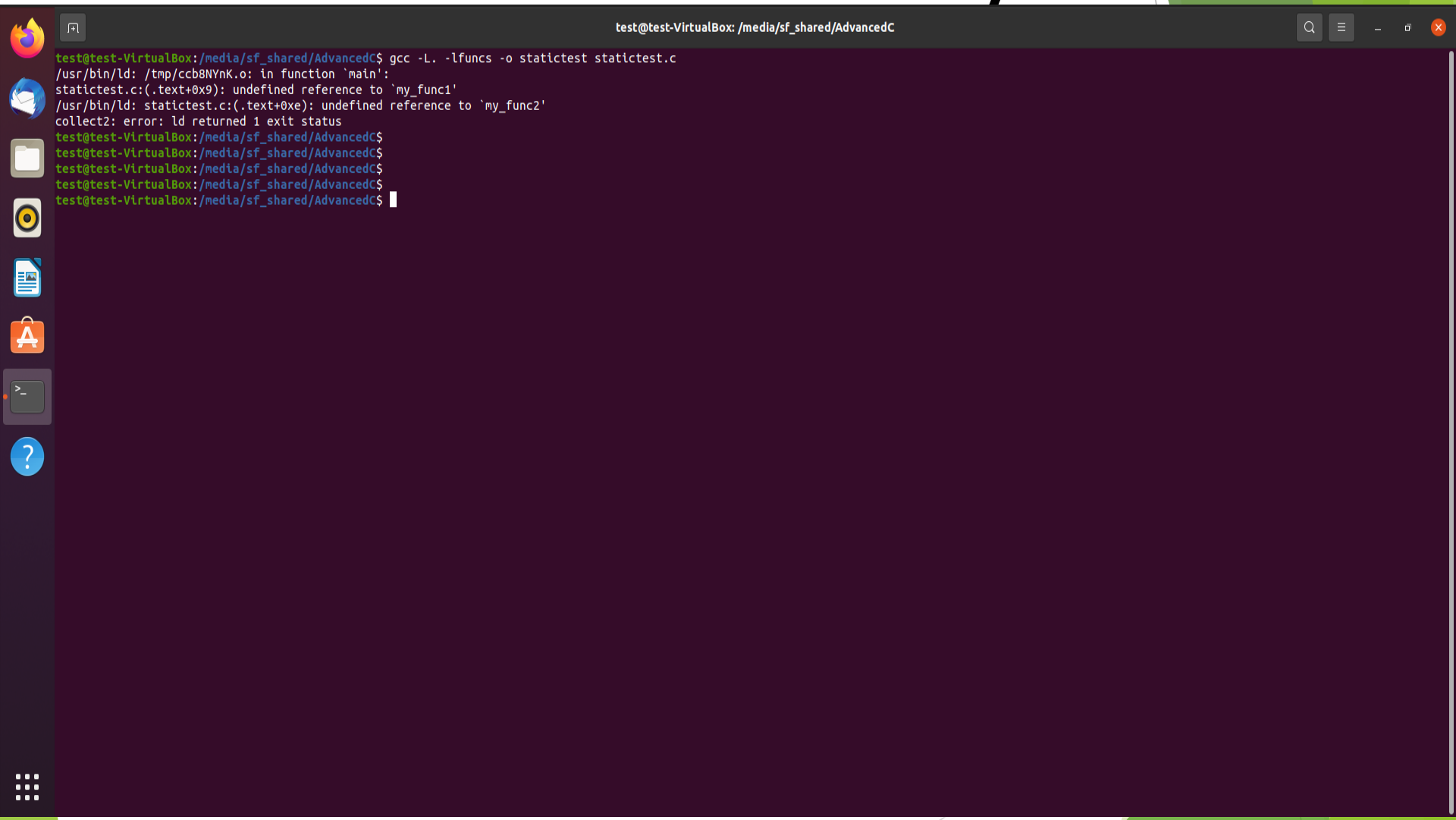
Static Library

- If linker find the reference, it will extract the object from archive and put it in our exe.

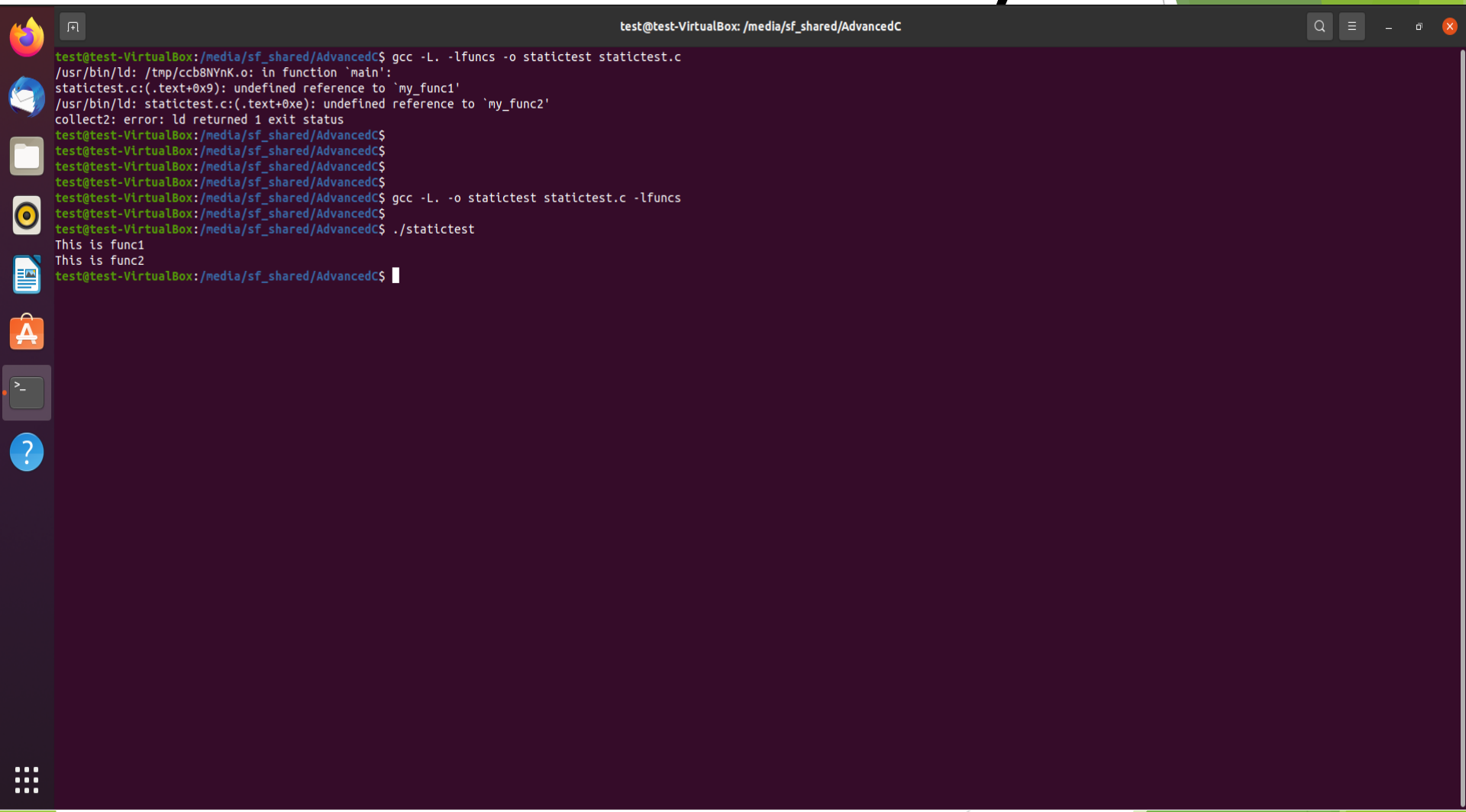
- If linker could not find any references, it shows an error and stops.

IT IS IMPORTANT TO PASS THE COMMAND LINE OPTIONS IN CORRECT ORDER

Static Library



Static Library



Shared Library

Shared Library

- Is also a collection of objects.

- When it is linked into another program, the program does not contain the whole objects, but just references to the shared library.

- Is not a collection of object files, but a single big object file which is a combination of object files.

- Shared Libraries are Position Independent Codes, because the function in a SO, may be loaded at different addresses in different programs.

Shared Library

Shared Library

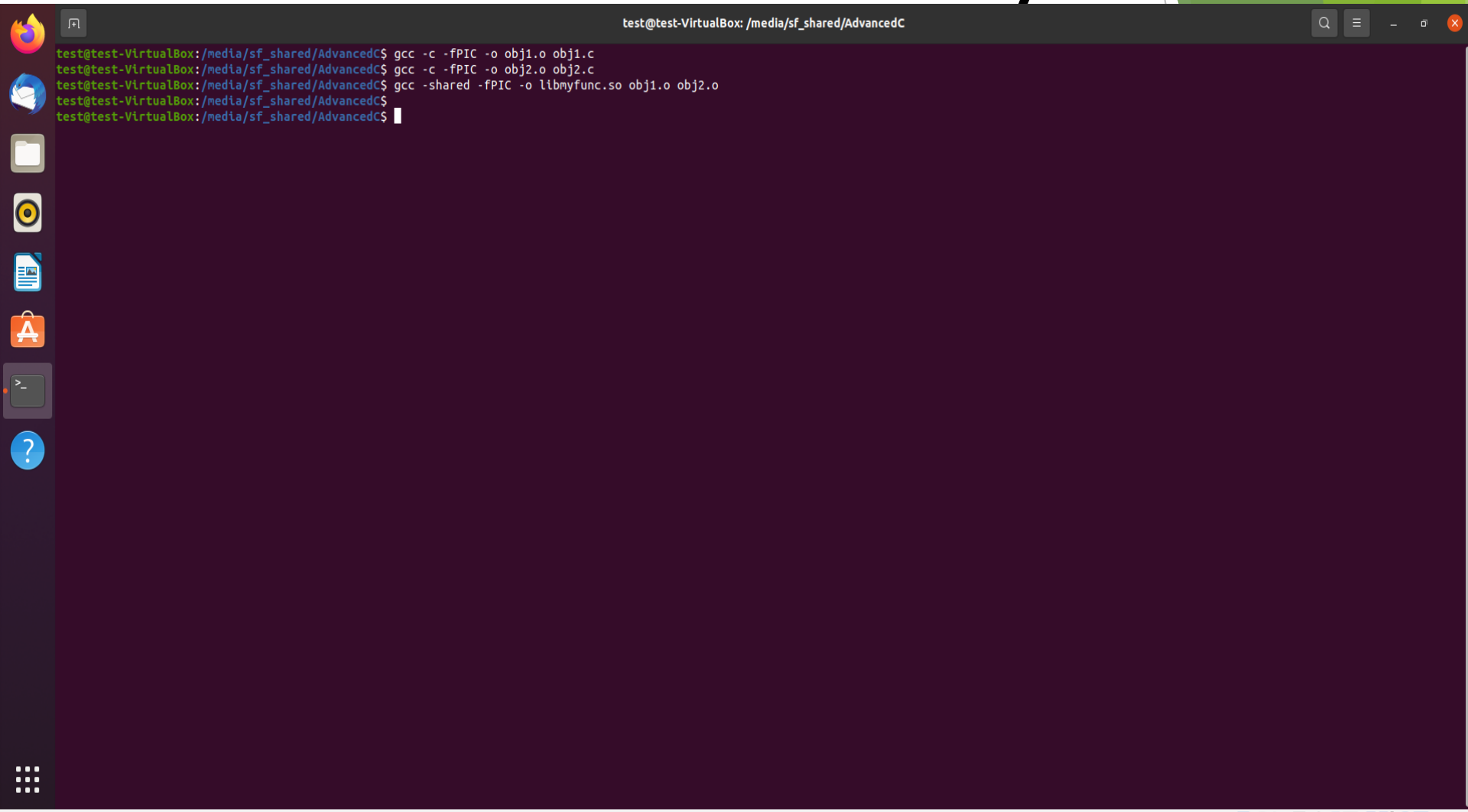
- The linker just includes the name of the “so” in executable file.

- The Operating System is responsible to find the specified “so” file.

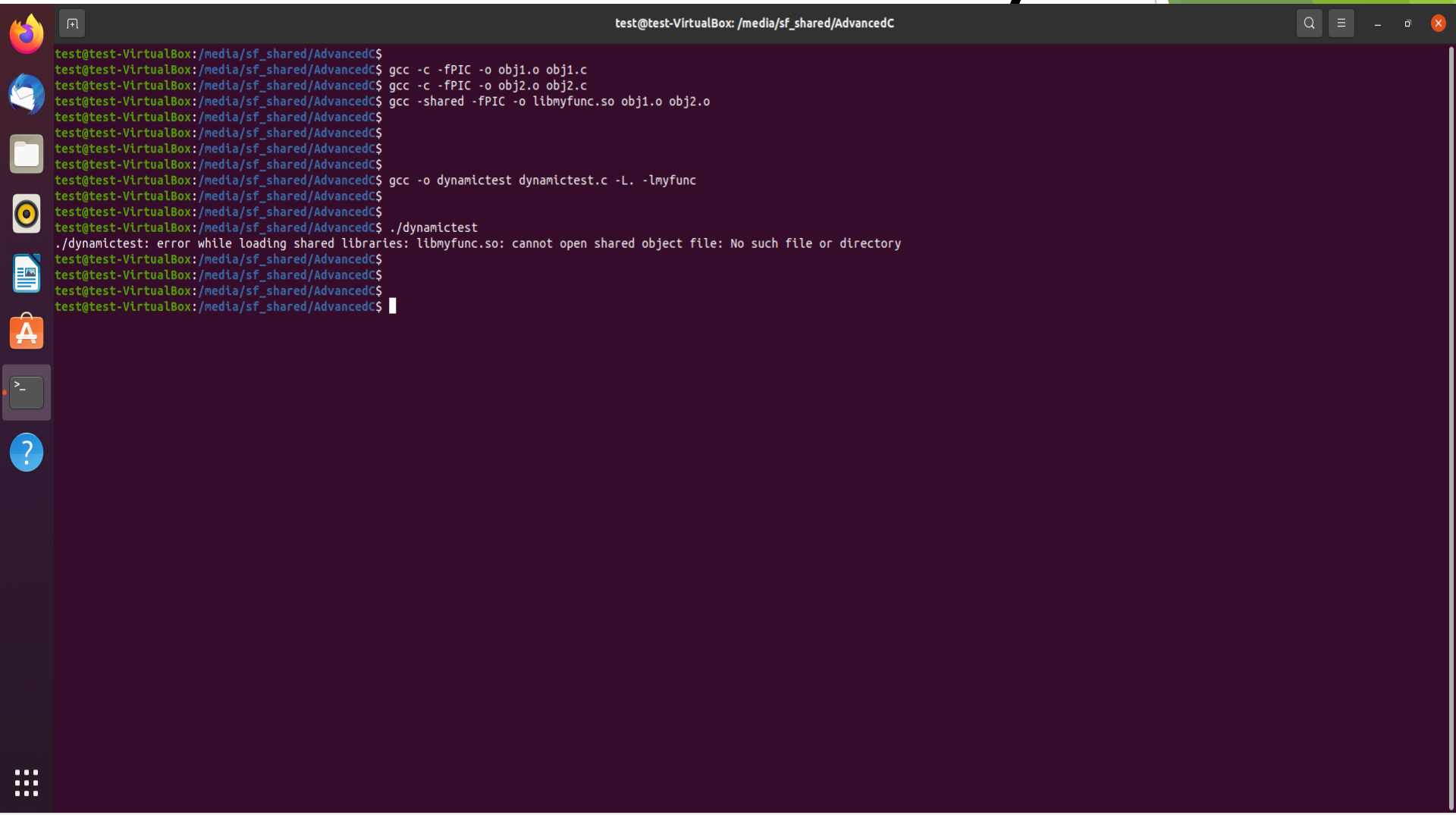
- By default, system searches only “/lib” and “/usr/lib”.

- You can indicate another path by setting the LD_LIBRARY_PATH environment variable.

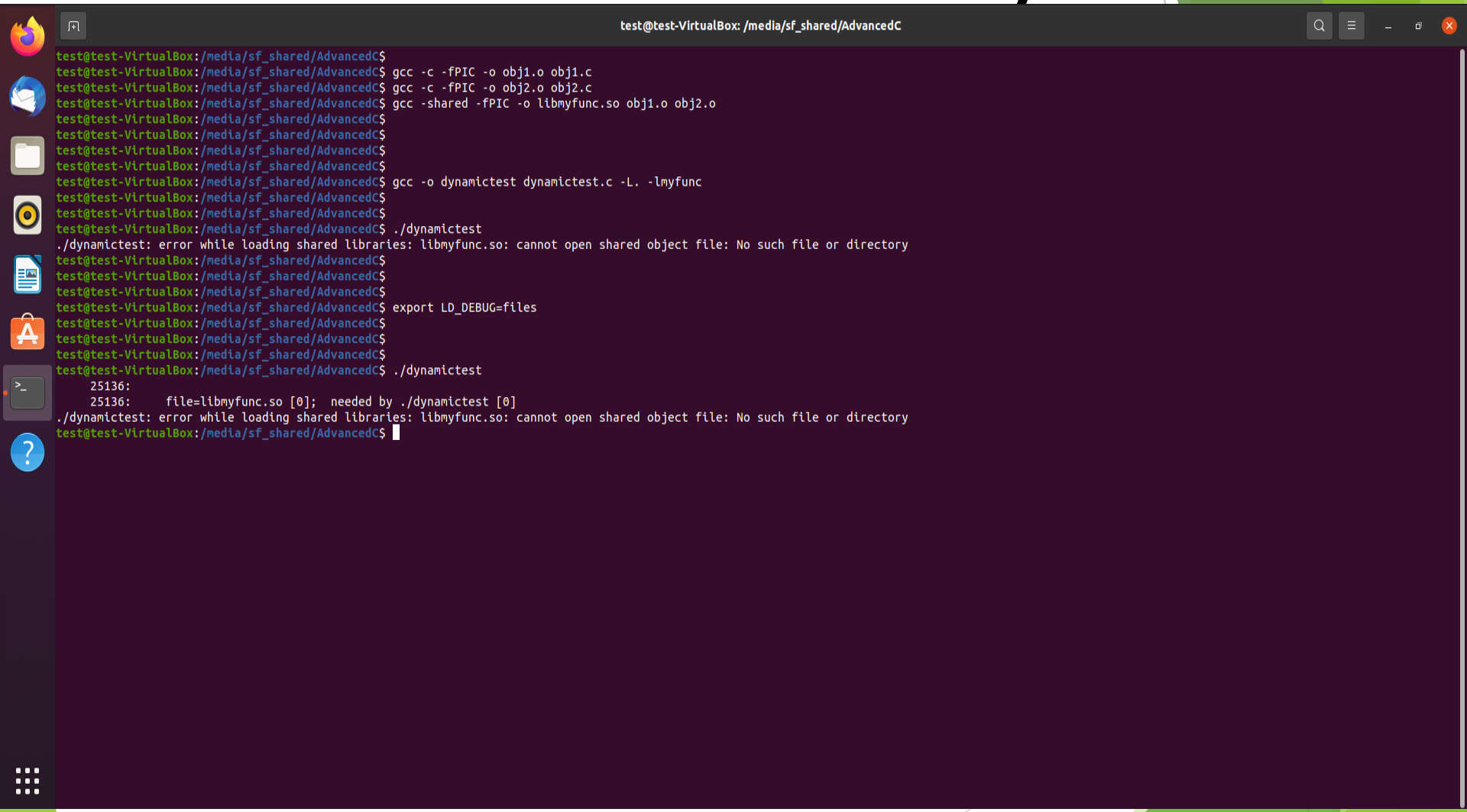
Shared Library



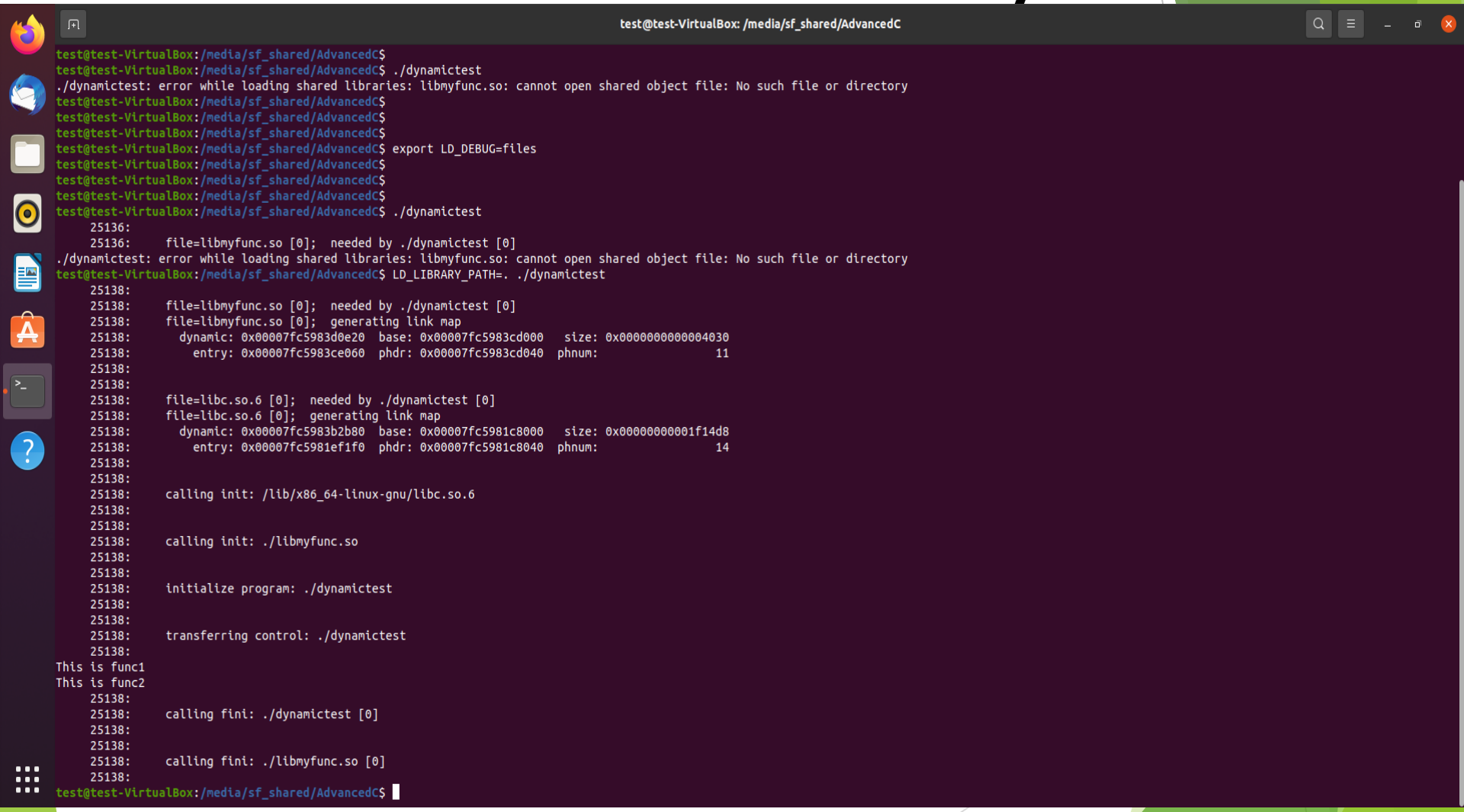
Shared Library



Shared Library



Shared Library



Shared Library

Libraries in Linux

- The ldd command shows the shared libraries that are linked into an executable (and their dependencies).

- Static libs, can not point to another lib, so you should include all dependent libs in GCC command line.

- The included SO s, need to be available during execution.

- The linker will stop searching for libraries when it finds a directory containing the proper ".so" or ".a".

- Priority of ".so" is higher than ".a" unless explicitly specified (-static option in gcc).

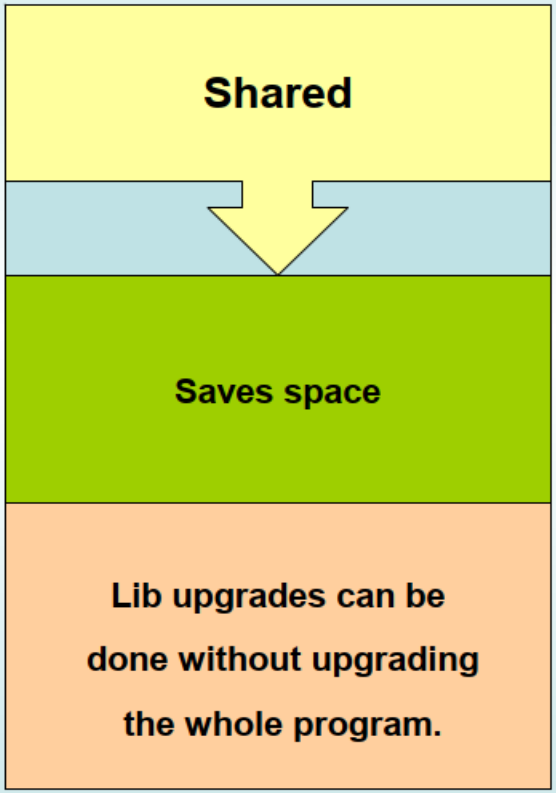
Shared Library

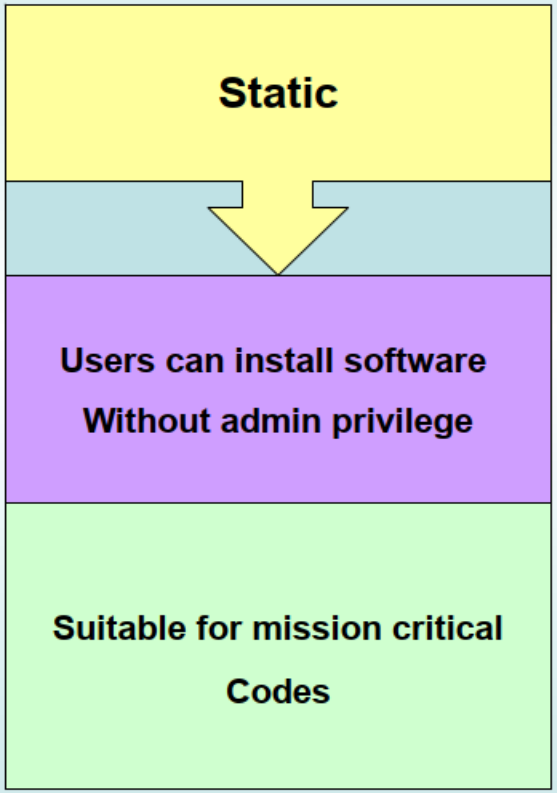
Shared Library

While compiling a source, you should indicate which libraries (which .SO) are needed.

While executing the code, the indicated SO should be available otherwise the code will not run.

Shared vs. Static





Error Handling

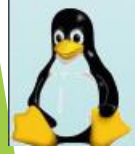
Error Handling

- A program may encounter a situation which can not work correctly.

- In case of an error, your program may decide to:

  - Ignore the error and continue running.
  - Stop working immediately.
  - Decide what to do next (is error recoverable?)

- The ability of a program to deal with errors is called "*Error Handling*".



Error Handling

- The first step in handling an error is to determine it's happened.

- In your program, you are responsible of checking for errors.

- In Linux, when calling a system call, if some error happens, the system call will set the *errno* global variable.

- Most system calls, return -1 on error and set errno respectively.

- After performing any system call, it's up to you to check the return value of a call and deal with probable errors.

Error Handling

- The *errno* variable is global, so you should check it exactly after desired call.

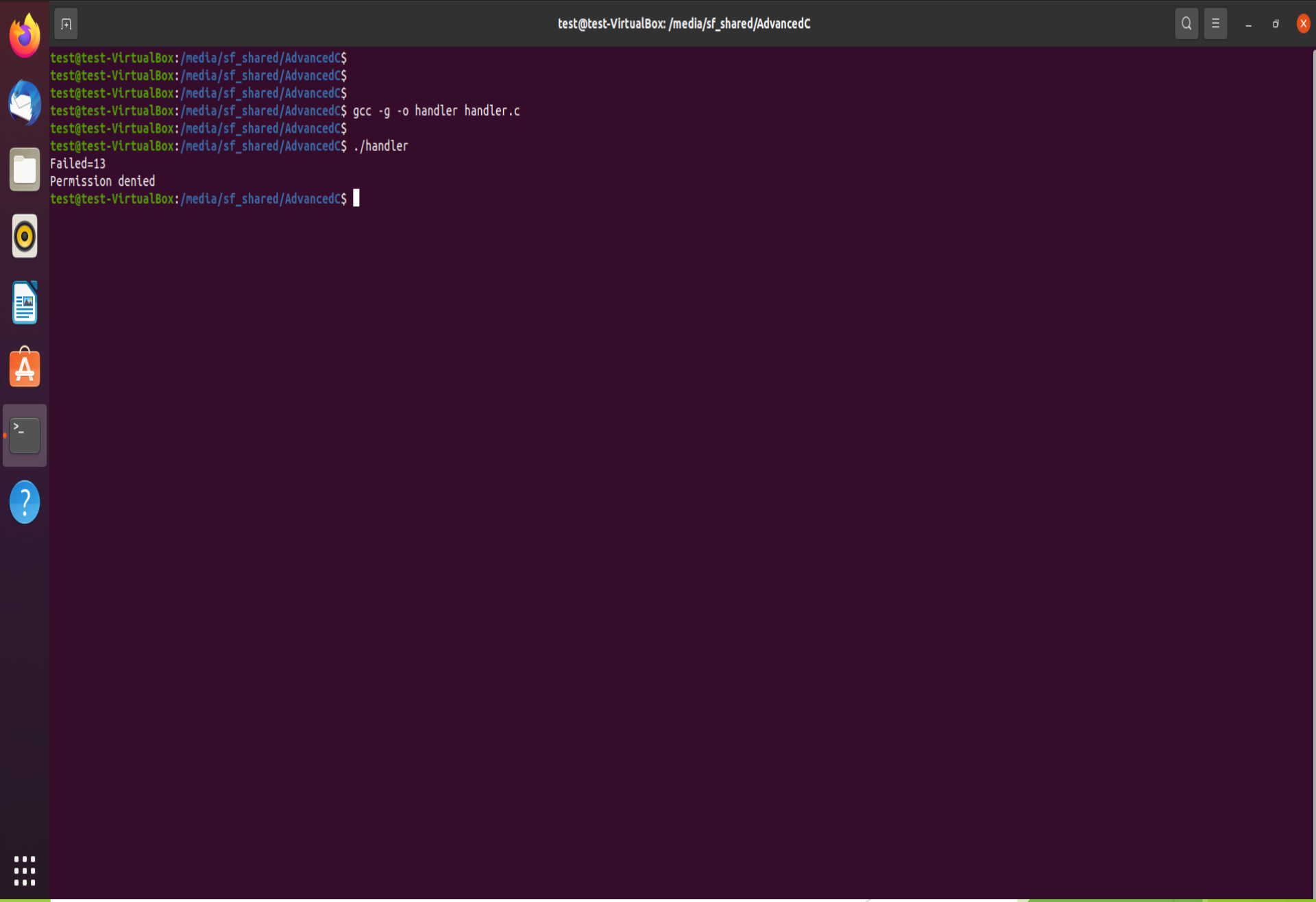
- *errno* is thread-safe.

- There are some functions to work with errno and print meaningful error messages.

- Using *strerror()* and *strerror_r()* is an option to deal with errors.

- These functions will return a string describing the error code passed in the argument.

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

int main()
{
FILE* fd;
fd=fopen("/etc/shadow","w");
if(fd==NULL)
{
        printf("Failed=%d\n",errno);
        printf("%s\n",strerror(errno));
}
else
{
        fclose(fd);
}
return 0;
}
```


test@test-VirtualBox: /media/sf_shared/AdvancedC
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -g -o handler handler.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./handler
Failed=13
Permission denied
test@test-VirtualBox:/media/sf_shared/AdvancedC$

Table

number	hex	symbol	description
1	0x01	EPERM	Operation not permitted
2	0x02	ENOENT	No such file or directory
3	0x03	ESRCH	No such process
4	0x04	EINTR	Interrupted system call
5	0x05	EIO	Input/output error
6	0x06	ENXIO	No such device or address
7	0x07	E2BIG	Argument list too long
8	0x08	ENOEXEC	Exec format error
9	0x09	EBADF	Bad file descriptor
10	0x0a	ECHILD	No child processes
11	0x0b	EAGAIN	Resource temporarily unavailable
11	0x0b	EWOULDBLOCK	*(Same value as EAGAIN)* Resource temporarily unavailable
12	0x0c	ENOMEM	Cannot allocate memory
13	0x0d	EACCES	Permission denied
14	0x0e	EFAULT	Bad address
15	0x0f	ENOTBLK	Block device required
16	0x10	EBUSY	Device or resource busy
17	0x11	EEXIST	File exists
18	0x12	EXDEV	Invalid cross-device link
19	0x13	ENODEV	No such device

https://chromium.googlesource.com/chromiumos/docs/+/master/constants/errnos.md

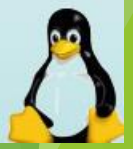
Error Handling

- You may also use the *assert* macro in your C program.

- One may use *assert* to properly generate some information in case of unpredicted situations.

- You can disable all *assert* s in your code providing – DNDEBUG option in your gcc command line.

- You should never perform any operation in your *assert* statement. Just check it.



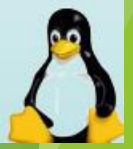
Error Handling

- You may also use the *assert* macro in your C program.

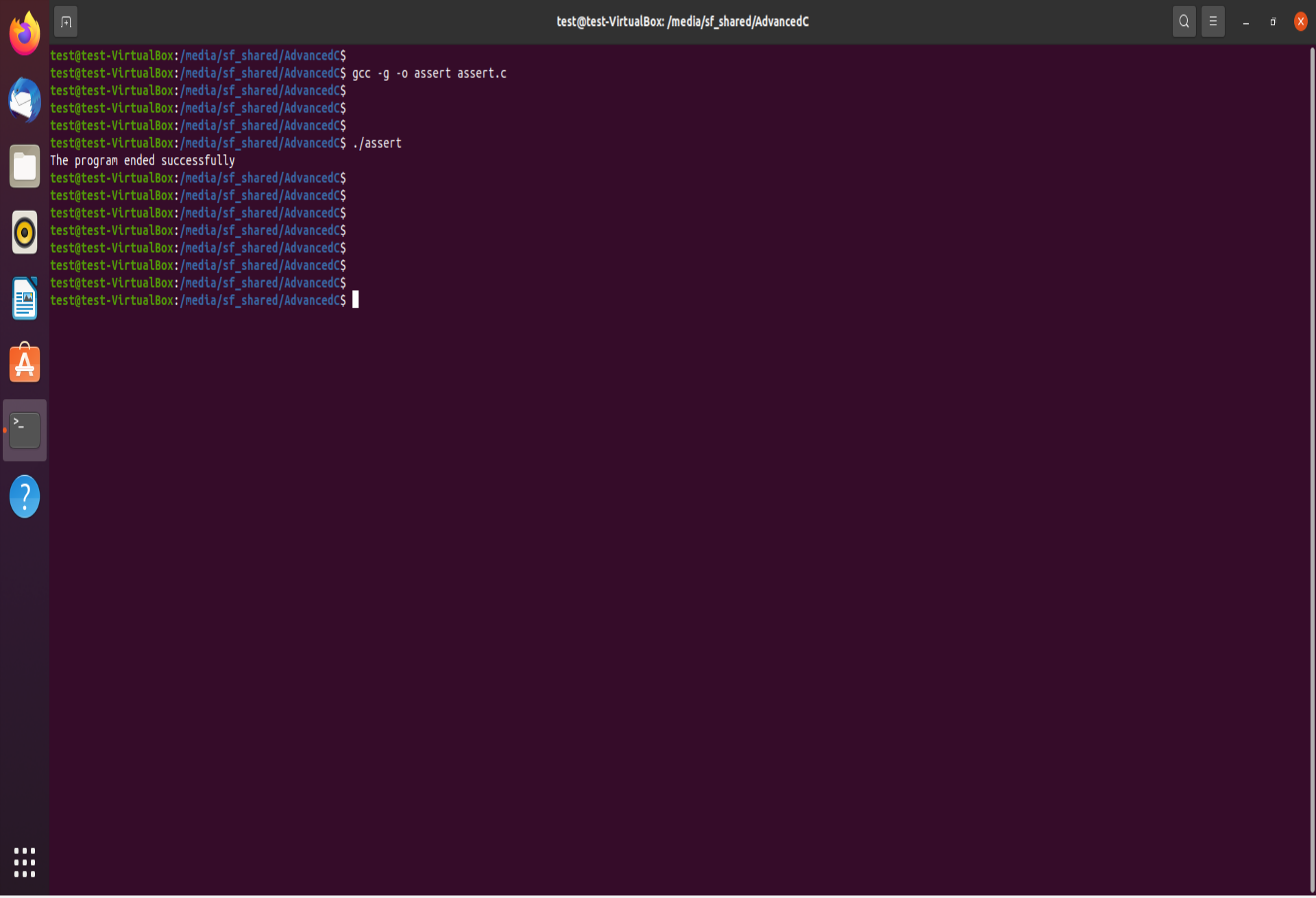
- One may use *assert* to properly generate some information in case of unpredicted situations.

- You can disable all *assert* s in your code providing – DNDEBUG option in your gcc command line.

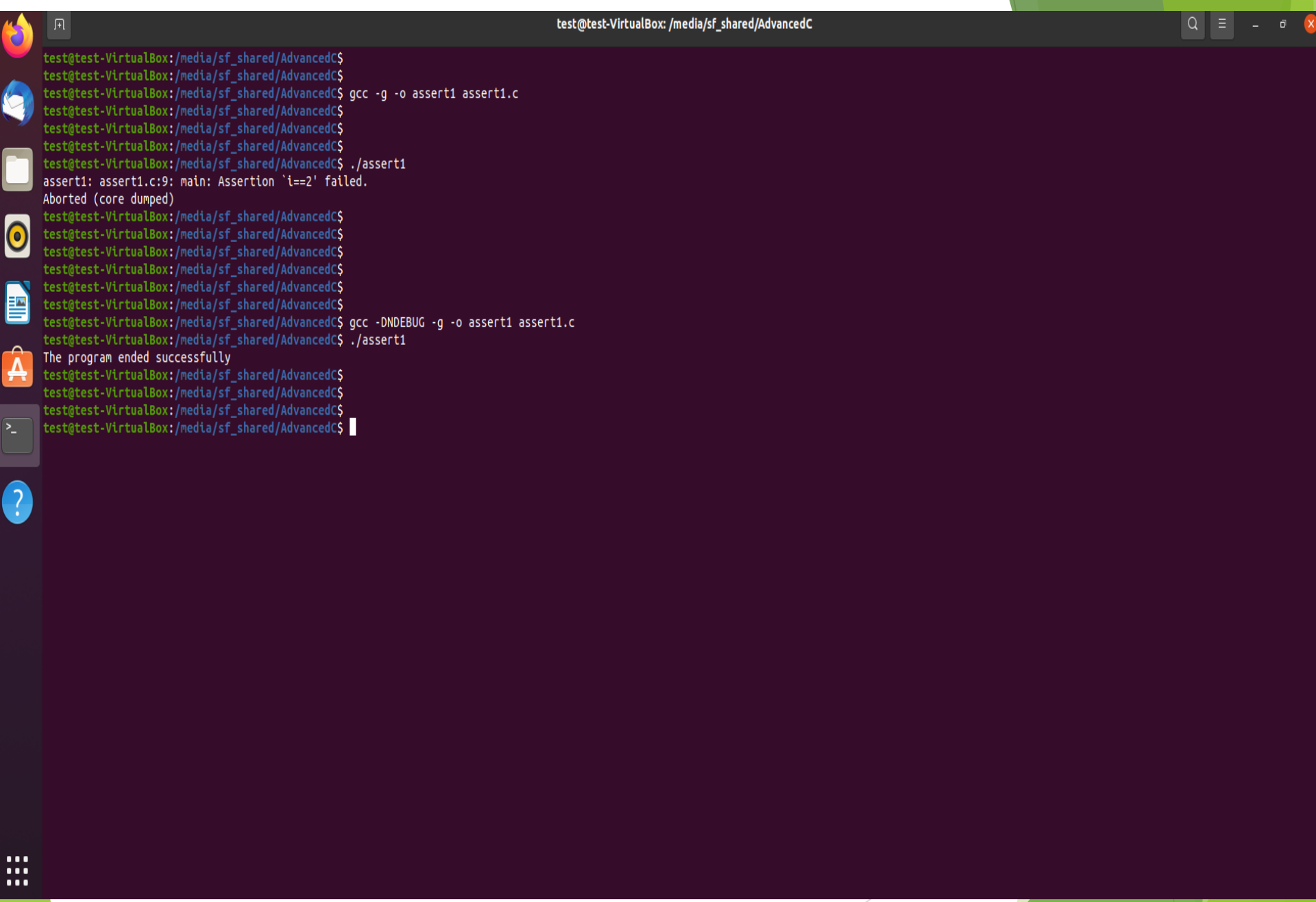
- You should never perform any operation in your *assert* statement. Just check it.



```
#include<stdio.h>
#include<assert.h>

int main()
{
int i;
i=2;
assert(i==2);
printf("The program ended successfully\n");
return 0;
}
```


test@test-VirtualBox: /media/sf_shared/AdvancedC
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -g -o assert assert.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./assert
The program ended successfully
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$

```
#include<stdio.h>
#include<assert.h>

int main()
{
int i;
i=2;
i=1;
assert(i==2);
printf("The program ended successfully\n");
return 0;
}
```


test@test-VirtualBox: /media/sf_shared/AdvancedC
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -g -o assert1 assert1.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./assert1
assert1: assert1.c:9: main: Assertion `i==2' failed.
Aborted (core dumped)
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -DNDEBUG -g -o assert1 assert1.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./assert1
The program ended successfully
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$

Open function

Open and possibly create a file or device

```
#include <sys/types.h>
    #include <sys/stat.h>
    #include <fcntl.h>
    int open(const char *pathname, int flags);
    int open(const char *pathname, int flags, mode_t mode);
    int creat(const char *pathname, mode_t mode);
```

DESCRIPTION

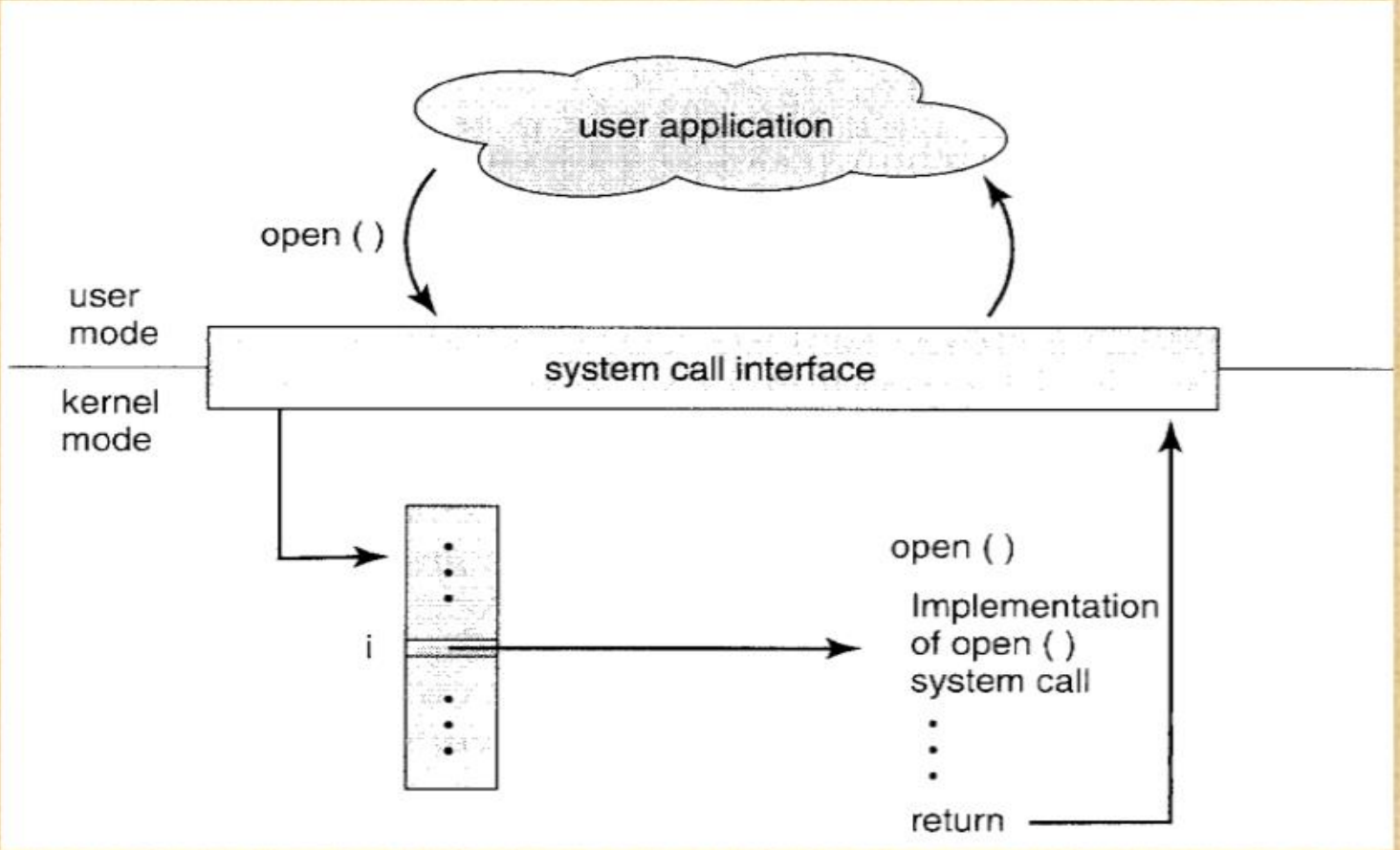
Given a pathname for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (read, write, lseek, fcntl, etc.).

Flags : One of the access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.

RETURN VALUE

open() and creat() return the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

Handling open system call



Close function

Close a file descriptor

SYNOPSIS

```
#include <unistd.h>
int close(int fd);
```

DESCRIPTION

close() closes a file descriptor, so that it no longer refers to any file and may be reused.

RETURN VALUE

close() returns zero on success. On error, -1 is returned, and errno is set appropriately.

Read function

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 descriptor fd into the buffer starting at buf.

If count is zero, read() returns zero and has no other results. If count is greater than SSIZE_MAX, the result is

unspecified.

RETURN VALUE

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. On error, -1 is returned, and errno is set appropriately.

Write function

Write to a file descriptor

SYNOPSIS

```
#include <unistd.h>

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

DESCRIPTION

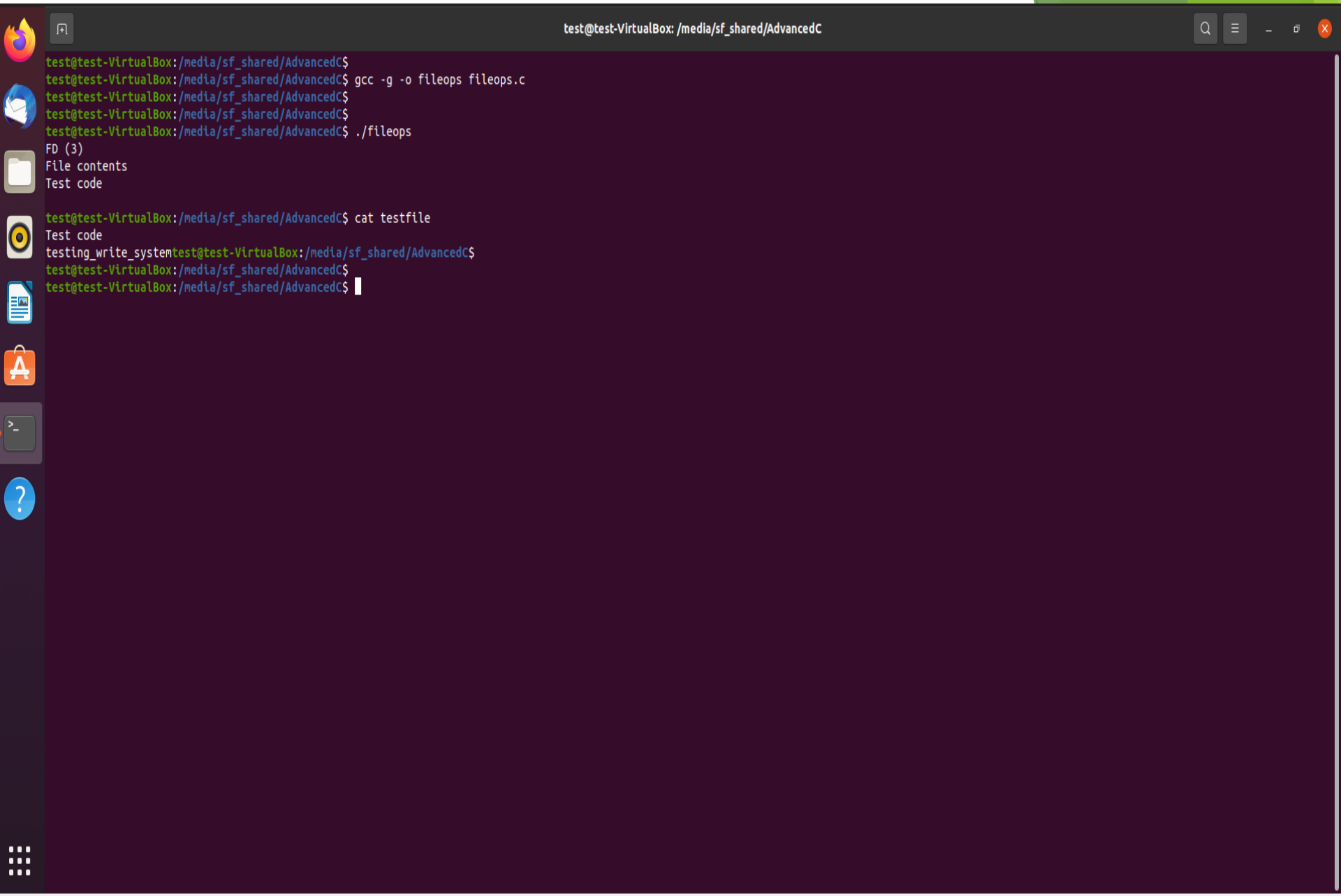
write() writes up to count bytes from the buffer pointed buf to the file referred to by the file descriptor fd.

RETURN VALUE

On success, the number of bytes written is returned (zero indicates nothing was written). On error, -1 is returned, and errno is set appropriately.

```
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main (){
 int fd, read_bytes;
 char buff[256];
 fd = open("testfile",O_RDWR);
 if (fd < 0){
                printf("Error opening file\n");
                return 0;
 }
 printf("FD (%d)\n", fd);
 read_bytes = read(fd, buff, 15);
 if (read_bytes < 0){
                printf("Read Error\n");
                close(fd);
                return 0;
 }
 printf("File contents\n");
 printf("%s\n", buff);
 strcpy(buff,"testing_write_system_call");
 read_bytes = write(fd, buff, 20);
 close(fd);
}
```


test@test-VirtualBox: /media/sf_shared/AdvancedC
Test code
"testfile" 1L, 10C
1,9
All


test@test-VirtualBox: /media/sf_shared/AdvancedC
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -g -o fileops fileops.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./fileops
FD (3)
File contents
Test code
test@test-VirtualBox:/media/sf_shared/AdvancedC$ cat testfile
Test code
testing_write_systemtest@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$

lseek function

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

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

SEEK_SET - The offset is set to offset bytes.

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

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

RETURN VALUE

Upon successful completion, lseek() returns the resulting offset location as measured in bytes from the beginning of the file. Otherwise, a value of (off_t) -1 is returned and errno is set to indicate the error.

access

- Testing file permissions

SYNOPSIS

- Access (const char *pathname, permission_flags)
- Permission flags - R_OK, W_OK, X_OK for read, write and execute permissions.

DESCRIPTION

The access system call determines whether the calling process has access permission to a file.

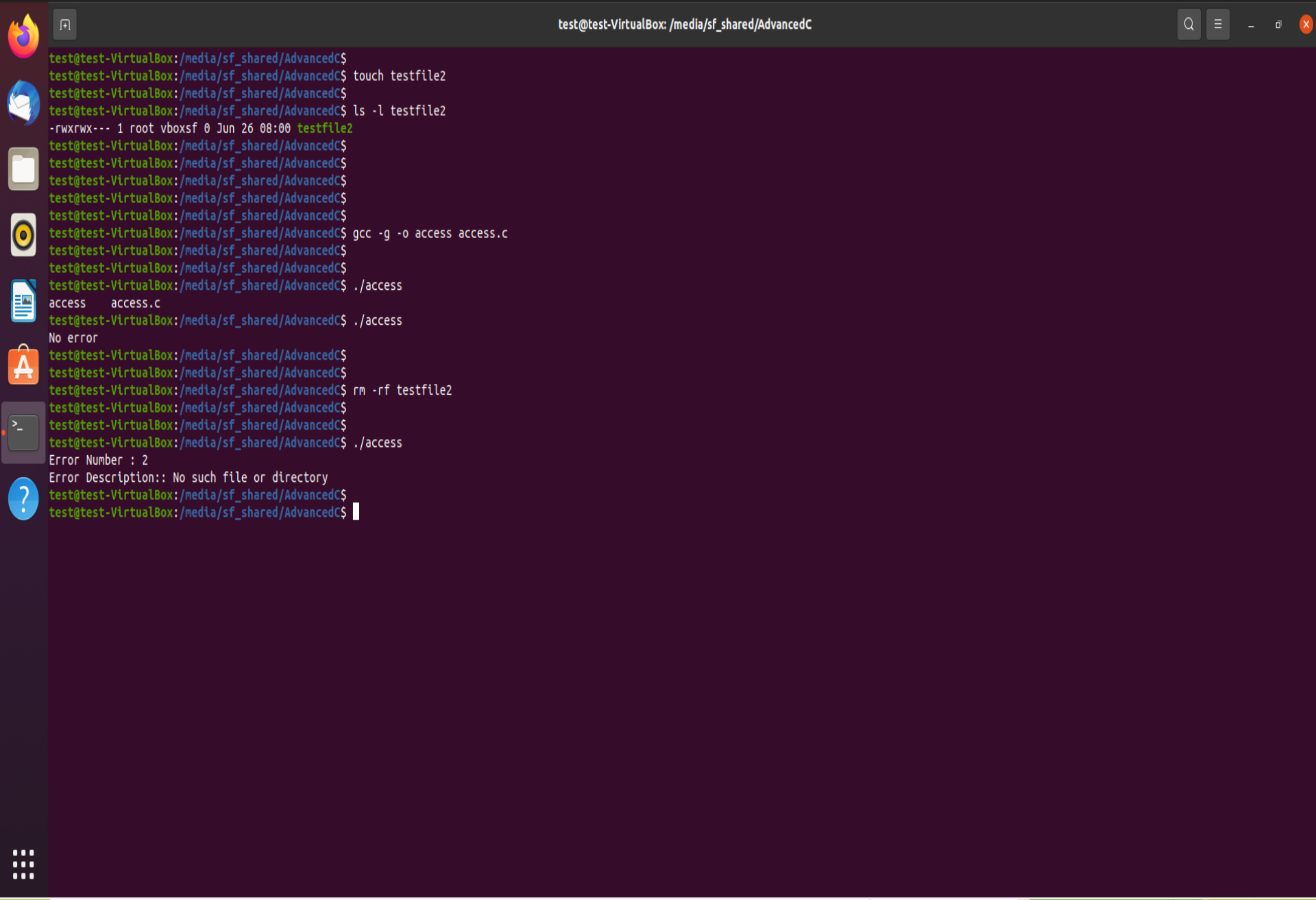
- It can check any combination of read (r), write (w) and execute (x) permission.
- It can also check whether file exists or not.

RETURN VALUE

On success return value is 0, if process has all specified permissions,

If the file exists and process does not have the specified permissions, access returns -1 and errno is set to EACCESS.

```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>


int main(int argc, const char *argv[]){
    int fd = access("testfile2", F_OK);
    if(fd == -1){
        printf("Error Number : %d\n", errno);
        perror("Error Description:");
    }
        else
                printf("No error\n");
    return 0;
}
```


test@test-VirtualBox: /media/sf_shared/AdvancedC
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ touch testfile2
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ls -l testfile2
-rwxrwx--- 1 root vboxsf 0 Jun 26 08:00 testfile2
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -g -o access access.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./access
access access.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./access
No error
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ rm -rf testfile2
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./access
Error Number : 2
Error Description:: No such file or directory
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$

dup and dup2 Functions:

- An existing file descriptor is duplicated by either of the following functions.

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2);
```

Both return: new file descriptor if OK,-1 on error.

- The new file descriptor returned by dup is guaranteed to be the lowest-numbered available file descriptor.

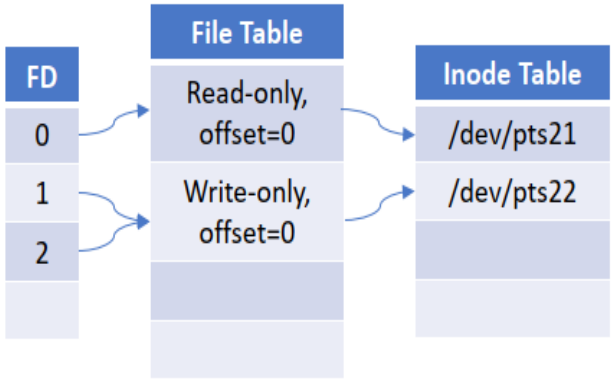
- With dup2, we specify the value of the new descriptor with the filedes2 argument. If filedes2 is already open, it is first closed. If filedes equals filedes2, then dup2 returns filedes2 without closing it.
- dup(filedes); is equivalent to

  fcntl(filedes, F_DUPFD, 0);
- dup2(filedes, filedes2); is equivalent to close(filedes2);

  fcntl(filedes, F_DUPFD, filedes2);

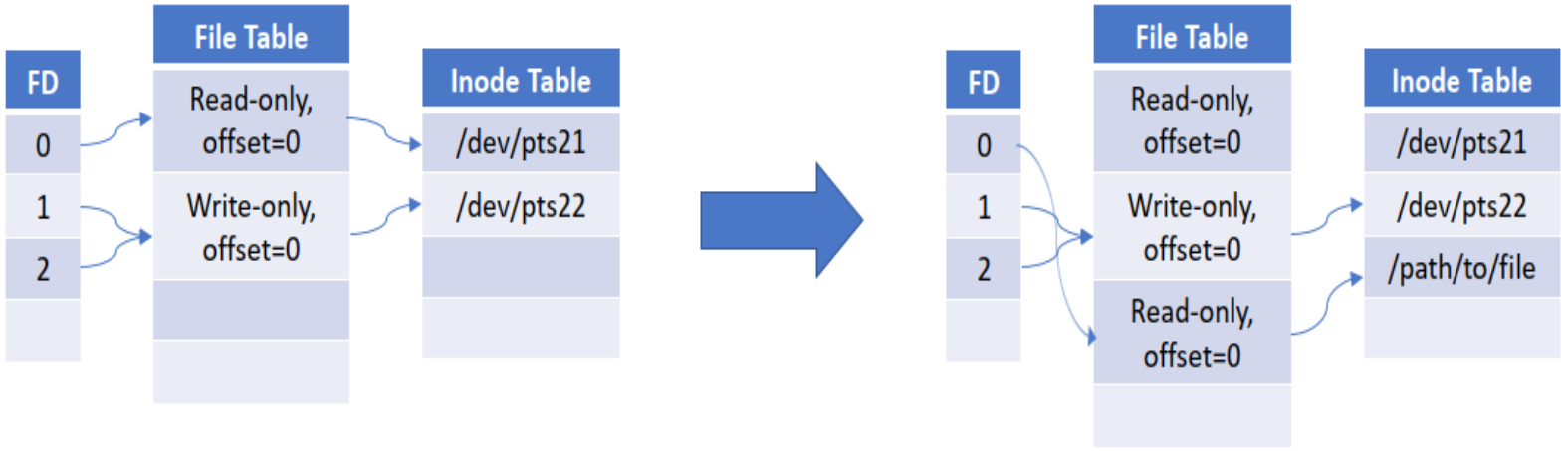
Example: Input Redirection

- Goal: instead of reading from stdin, read from another source



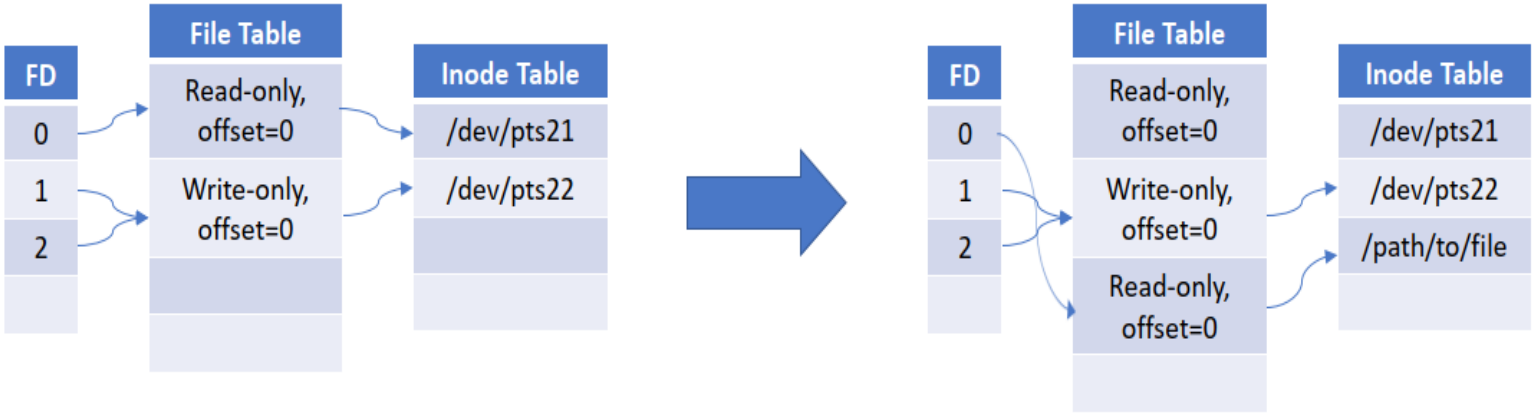
Example: Input Redirection

- Goal: instead of reading from stdin, read from another source



Example: Input Redirection

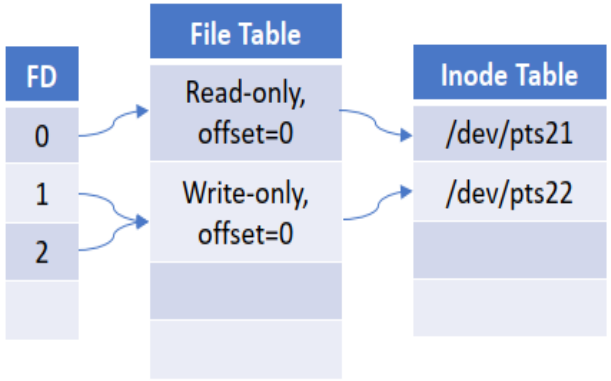
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

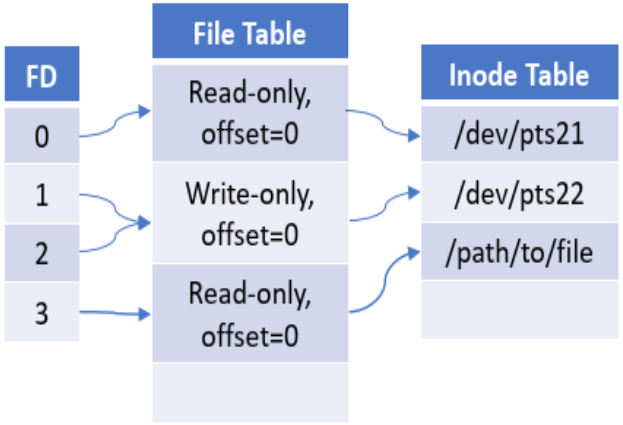
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

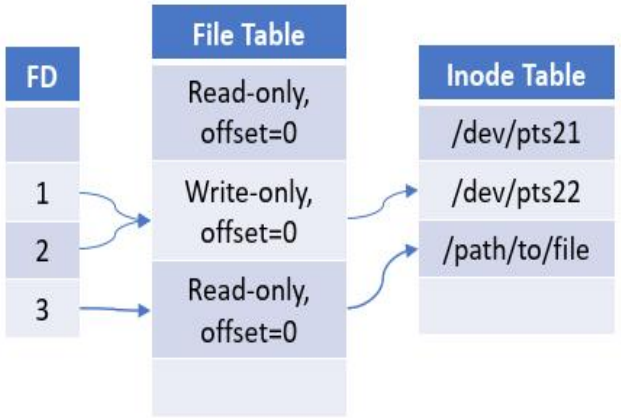
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

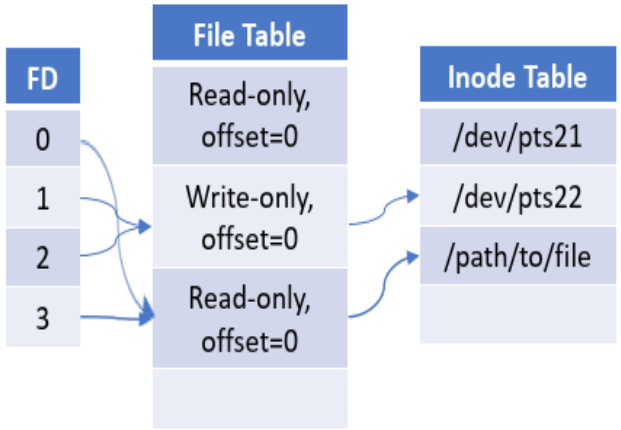
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

Example: Input Redirection

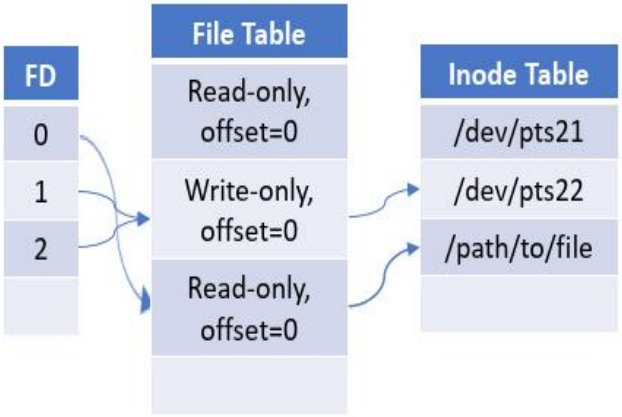
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

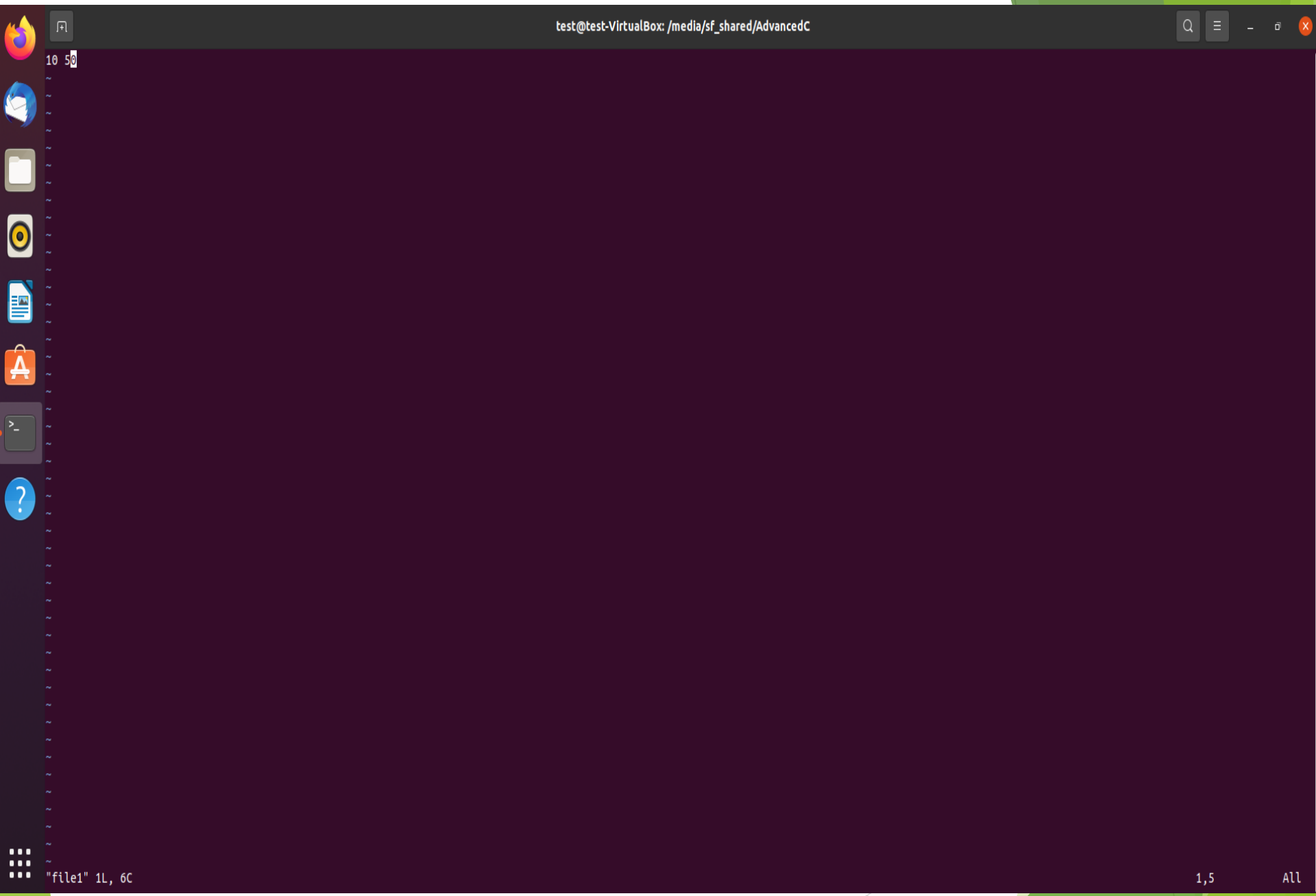
Example: Input Redirection

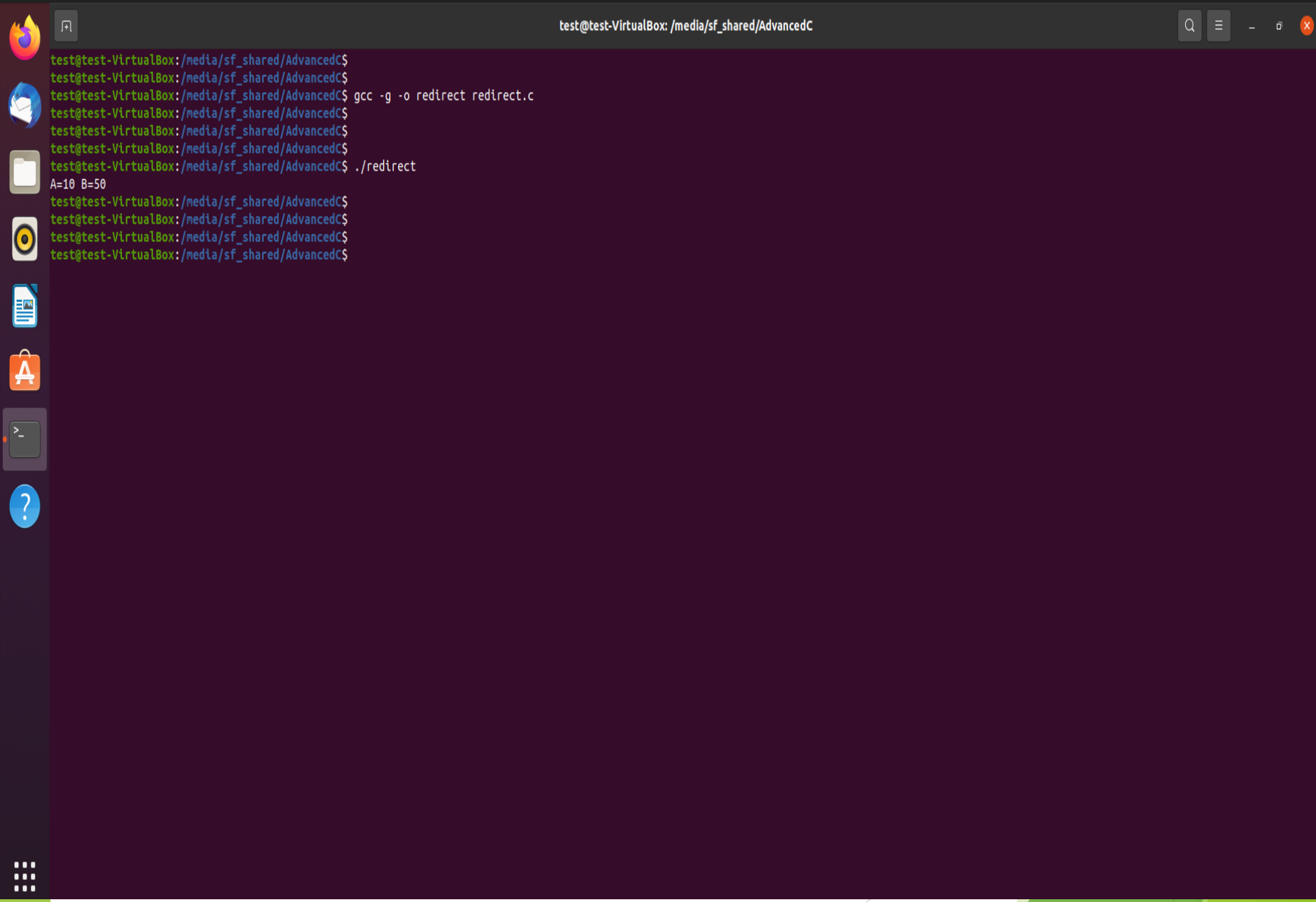
- Goal: instead of reading from stdin, read from another source




```
int ifd = open(newfile, O_RDONLY);
if (ifd >= 0) {
        close(0);
        dup(ifd);
        close(ifd);
}
```

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


int main()
{
int a,b;
int fd;
fd = open("file1",O_RDONLY);
if(fd>=0){
                close(0);
                dup(fd);
                close(fd);
}
scanf("%d %d",&a,&b);
printf("A=%d B=%d\n",a,b);
}
```


test@test-VirtualBox: /media/sf_shared/AdvancedC
10 50
"file1" 1L, 6C
1,5
All


test@test-VirtualBox: /media/sf_shared/AdvancedC
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -g -o redirect redirect.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./redirect
A=10 B=50
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$
test@test-VirtualBox:/media/sf_shared/AdvancedC$

MMAP Handson

Anandkumar

- FUSE is a loadable kernel module for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code.
- This is achieved by running file system code in user space while the FUSE module provides only a "**bridge**" to the actual kernel interfaces.

- FUSE is particularly useful for writing **virtual file systems**. Unlike traditional file systems that essentially save data to and retrieve data from disk, virtual filesystems do not actually store data themselves. They act as a view or translation of an existing file system or storage device.

- The FUSE system was originally part of A Virtual Filesystem (AVFS), but has since split off into its own project on SourceForge.net.

- FUSE is available for Linux, FreeBSD, NetBSD, OpenSolaris, and Mac OS X. It was officially merged into the mainstream Linux kernel tree in kernel version 2.6.14.

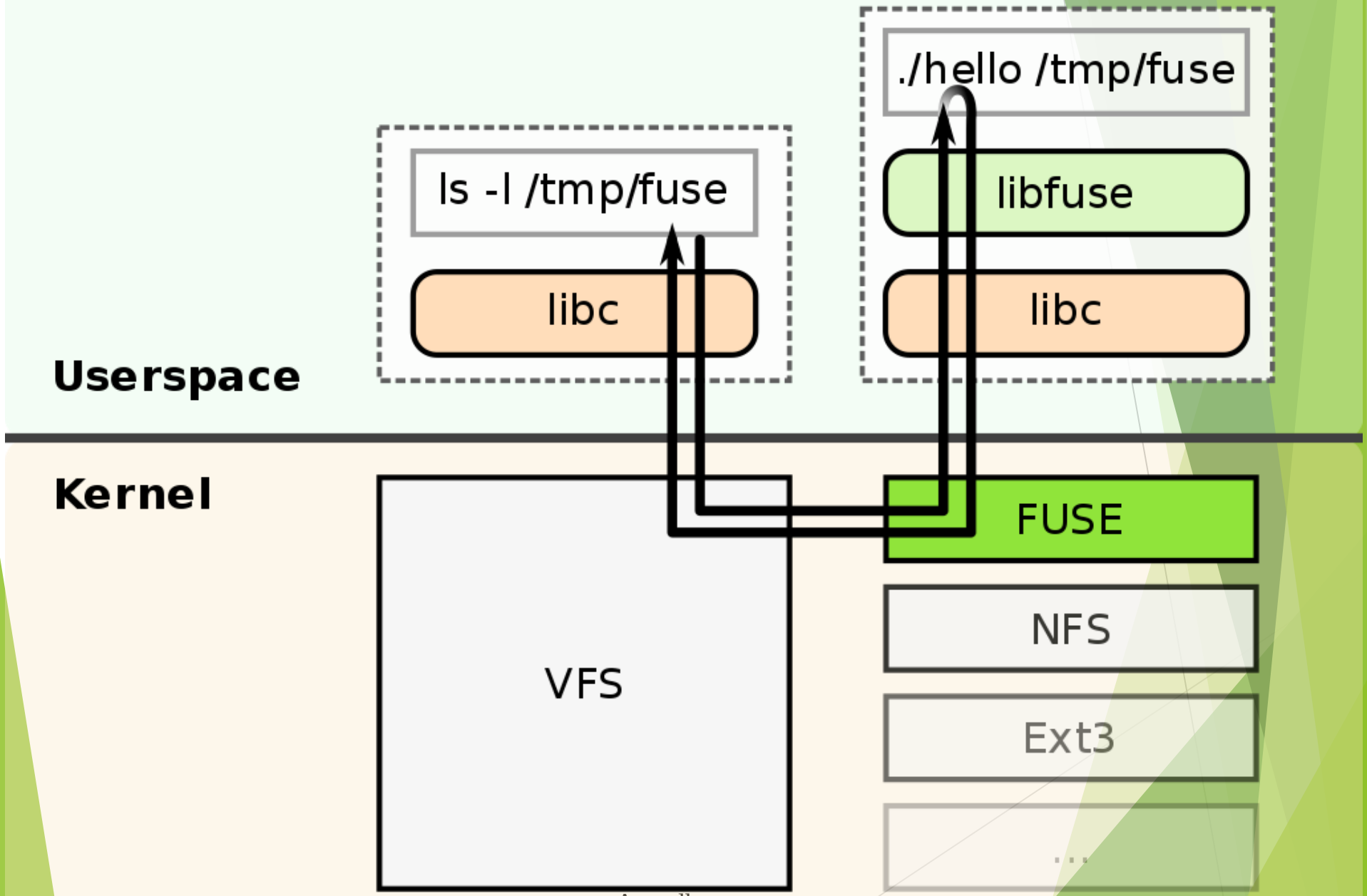
- **ExpanDrive**: A commercial filesystem implementing SFTP/FTP/FTPS using FUSE.
- **GlusterFS**: Clustered Distributed Filesystem having capability to scale up to several petabytes.
- **SSHFS**: Provides access to a remote filesystem through SSH.
- **GmailFS**: Filesystem which stores data as mail in Gmail
- **EncFS**: Encrypted virtual filesystem

FUSE structure

- FUSE kernel module (fuse.ko)
  - inode.c, dev.c, control.c, dir.c, file.c
- LibFUSE module (libfuse.*)
  - helper.c, fuse_kern_chan.c, fuse_mt.c, fuse.c, fuse_lowlevel.c, fuse_loop.c, fuse_loop_mt.c, fuse_session.c
- Mount utility(fusermount)
  - fusermount, mount.fuse.c, mount_util.c, mount.c, mount_bsd.c,

FUSE Library

- include/fuse.h → the library interface of FUSE（HighLevel）
- include/fuse_common.h → common
- include/fuse_lowlevel.h → Lowlevel API
- include/fuse_opt.h → option parsing interface of FUSE


./hello /tmp/fuse
ls -l /tmp/fuse
libfuse
libc
libc
Userspace
Kernel
FUSE
VFS
NFS
Ext3
...

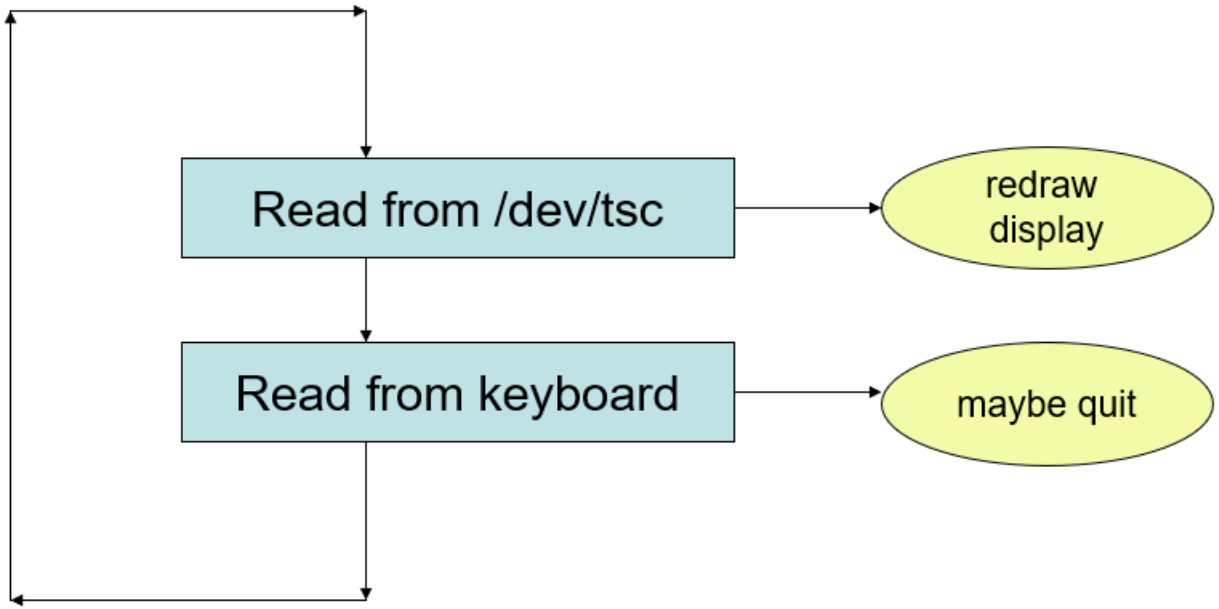
`getattr`	Get file attributes - similar to `stat()`.
`readlink`	Read the target of a symbolic link.
`mknod`	Create a file node.
`mkdir`	Create a directory.
`unlink`	Remove a file.
`rmdir`	Remove a directory.
`symlink`	Create a symbolic link.
`rename`	Rename a file.
`link`	Create a hard link to a file.
`chmod`	Change the permission bits of a file.

`chown`	Change the owner and group of a file.
`truncate`	Change the size of a file.
`open`	Open a file.
`release`	Close an open file.
`read`	Read data from an open file.
`write`	Write data to an open file.
`fsync`	Synchronize file contents (ie. flush dirty buffers).
`opendir`	Open directory.
`readdir`	Read directory - get directory listing.
`releasedir`	Close directory.

`utime`	Change the access and/or modification times of a file.
`statfs`	Get file system statistics.
`init`	Initialize filesystem (FUSE specific).
`destroy`	Clean up filesystem (FUSE specific).

https://github.com/libfuse/libfuse/blob/master/include/fuse.h

Illustrates I/O Multiplexing



More than just one source for input

Remember ‘read()’ semantics

If device-driver’s ‘read()’ function is called before the device has any data available, then the calling task is expected to ‘sleep’ on a wait-queue until the device has at least one byte of data ready to return.

The keyboard’s device-driver behaves in exactly that expected way: it ‘blocks’ if we try to read when there are no keystrokes.

Blocking versus Multiplexing

If we want to read from multiple devices, we are not able to do it properly with the read() system-call

If one device has no data, our task will be blocked, and so can’t read data that may become available from some other device

Idea: use ‘nonblocking’ i/o

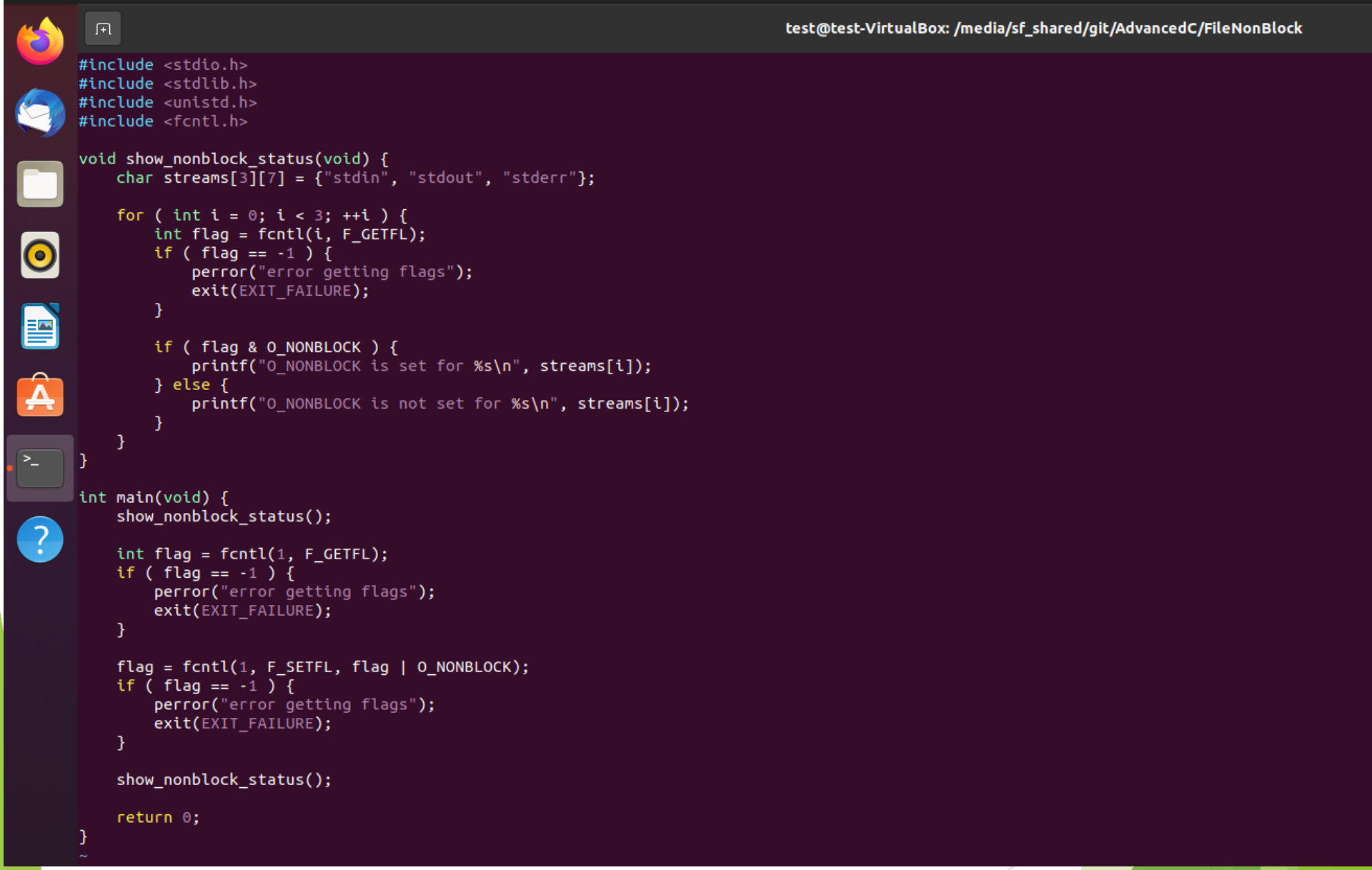
When we ‘open()’ a device-file, we could specify the ‘O_NONBLOCK’ i/o mode

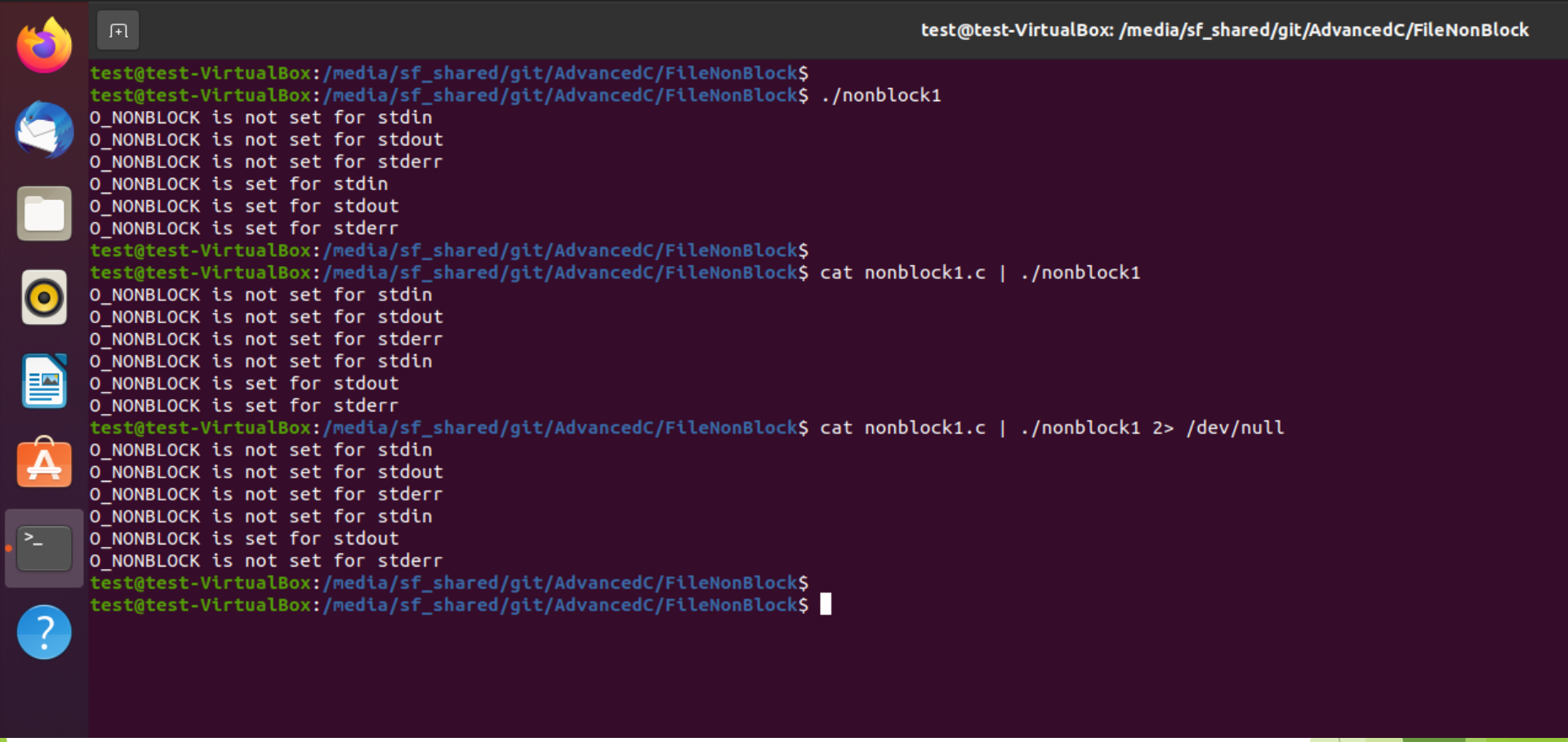
Idea: use ‘nonblocking’ i/o

When we ‘open()’ a device-file, we could specify the ‘O_NONBLOCK’ i/o mode

Idea: use ‘nonblocking’ i/o

When we ‘open()’ a device-file, we could specify the ‘O_NONBLOCK’ i/o mode


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/FileNonBlock
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
void show_nonblock_status(void) {
char streams[3][7] = {"stdin", "stdout", "stderr"};
for (int i = 0; i < 3; ++i) {
int flag = fcntl(i, F_GETFL);
if (flag == -1) {
perror("error getting flags");
exit(EXIT_FAILURE);
}
if (flag & O_NONBLOCK) {
printf("O_NONBLOCK is set for %s\n", streams[i]);
} else {
printf("O_NONBLOCK is not set for %s\n", streams[i]);
}
}
}
int main(void) {
show_nonblock_status();
int flag = fcntl(1, F_GETFL);
if (flag == -1) {
perror("error getting flags");
exit(EXIT_FAILURE);
}
flag = fcntl(1, F_SETFL, flag | O_NONBLOCK);
if (flag == -1) {
perror("error getting flags");
exit(EXIT_FAILURE);
}
show_nonblock_status();
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/FileNonBlock
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$ ./nonblock1
O_NONBLOCK is not set for stdin
O_NONBLOCK is not set for stdout
O_NONBLOCK is not set for stderr
O_NONBLOCK is set for stdin
O_NONBLOCK is set for stdout
O_NONBLOCK is set for stderr
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$ cat nonblock1.c | ./nonblock1
O_NONBLOCK is not set for stdin
O_NONBLOCK is not set for stdout
O_NONBLOCK is not set for stderr
O_NONBLOCK is not set for stdin
O_NONBLOCK is set for stdout
O_NONBLOCK is set for stderr
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$ cat nonblock1.c | ./nonblock1 2> /dev/null
O_NONBLOCK is not set for stdin
O_NONBLOCK is not set for stdout
O_NONBLOCK is not set for stderr
O_NONBLOCK is not set for stdin
O_NONBLOCK is set for stdout
O_NONBLOCK is not set for stderr
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$

```
char buf[BUFSIZ];
ssize_t nr;

start:
nr = read (fd, buf, BUFSIZ);
if (nr == -1) {
        if (errno == EINTR)
                goto start; /* oh shush */
        if (errno == EAGAIN)
                /* resubmit later */
        else
                /* error */
}
```

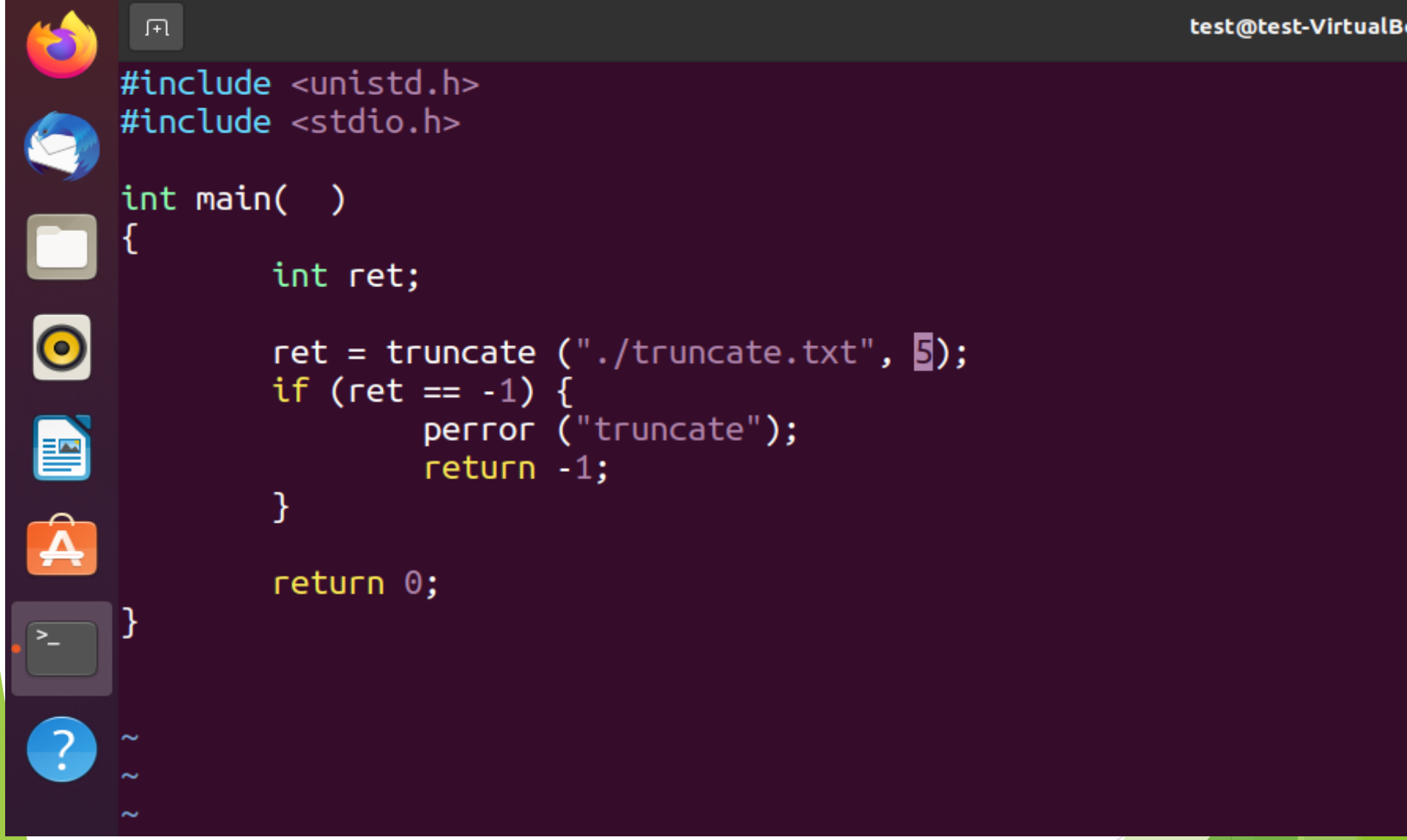
Linux provides two system calls for truncating the length of a file, both of which are defined and required (to varying degrees) by various POSIX standards. They are:

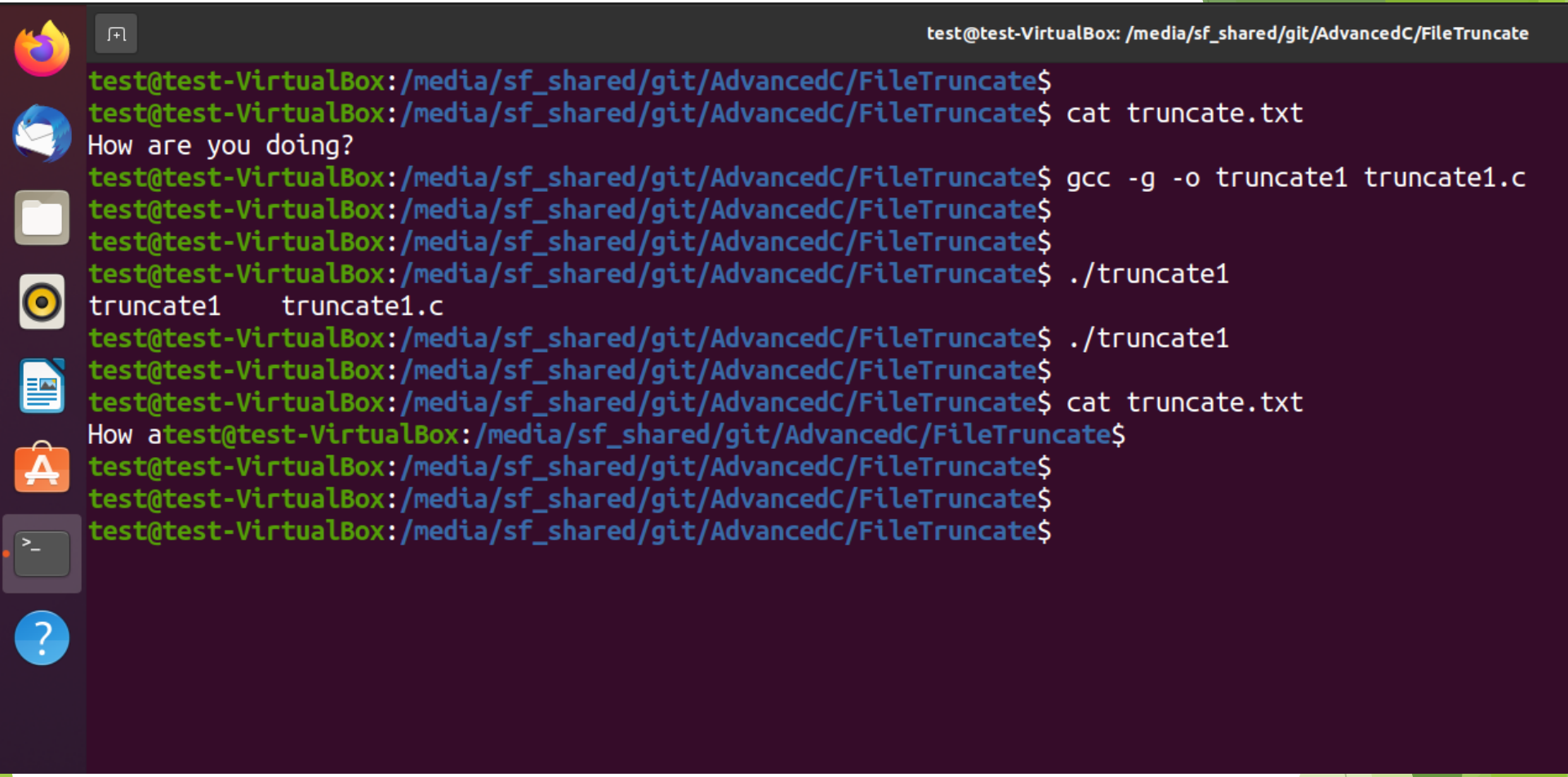
```
#include <unistd.h>
#include <sys/types.h>

int ftruncate (int fd, off_t len);
```

and:

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

int truncate (const char *path, off_t len);
```

Both system calls truncate the given file to the length given by `len`. The `ftruncate( )` system call operates on the file descriptor given by `fd`, which must be open for writing. The `truncate( )` system call operates on the filename given by `path`, which must be writable. Both return `0` on success. On error, they return `-1`, and set `errno` as appropriate.


test@test-VirtualB
#include <unistd.h>
#include <stdio.h>
int main( )
{
int ret;
ret = truncate ("./truncate.txt", 5);
if (ret == -1) {
perror ("truncate");
return -1;
}
return 0;
}
~
~
~


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/FileTruncate
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$ cat truncate.txt
How are you doing?
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$ gcc -g -o truncate1 truncate1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$ ./truncate1
truncate1 truncate1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$ ./truncate1
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$ cat truncate.txt
How atest@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileTruncate$

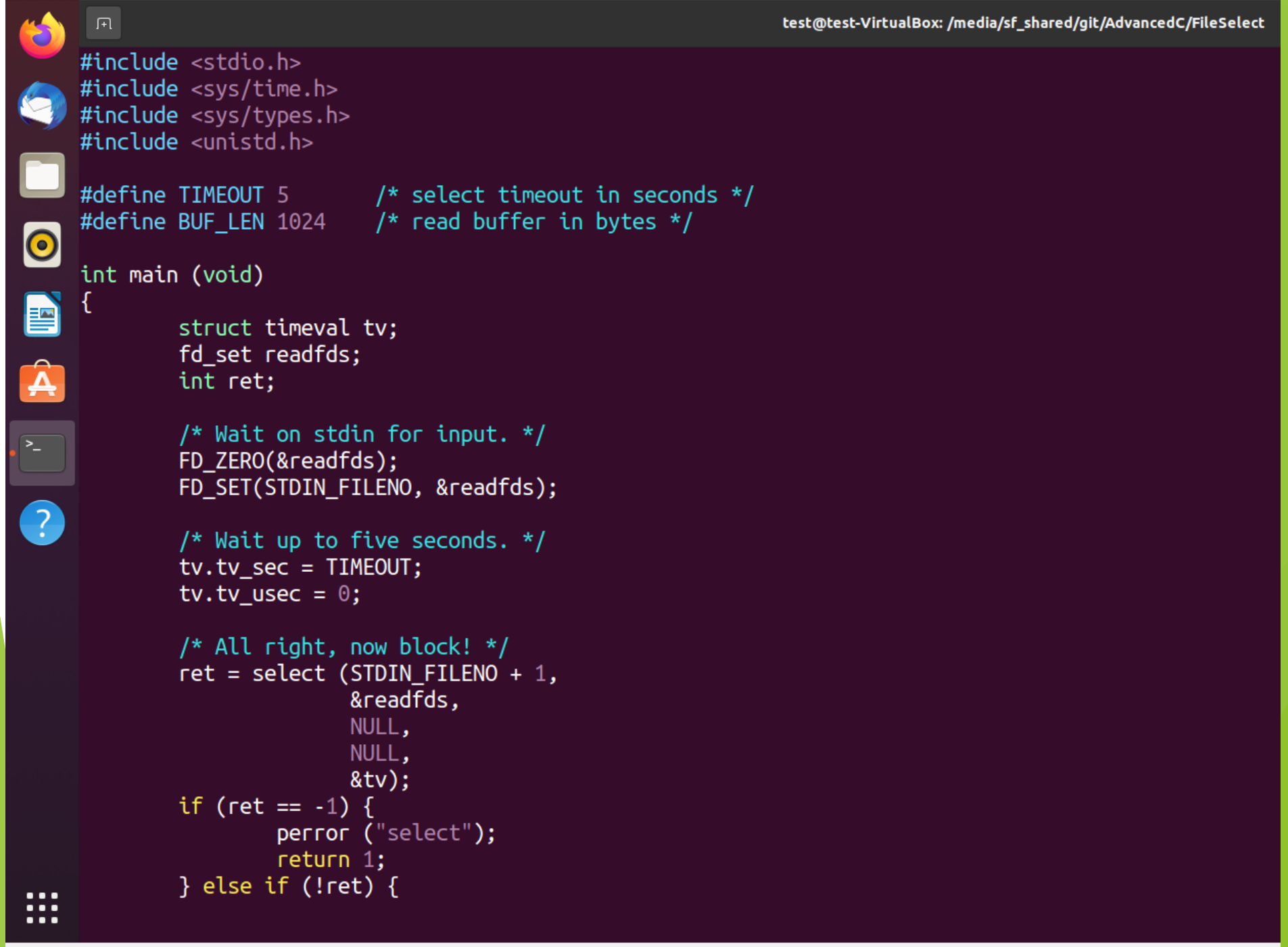
The `select( )` system call provides a mechanism for implementing synchronous multiplexing I/O:

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select (int n,
            fd_set *readfds,
            fd_set *writefds,
            fd_set *exceptfds,
            struct timeval *timeout);

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

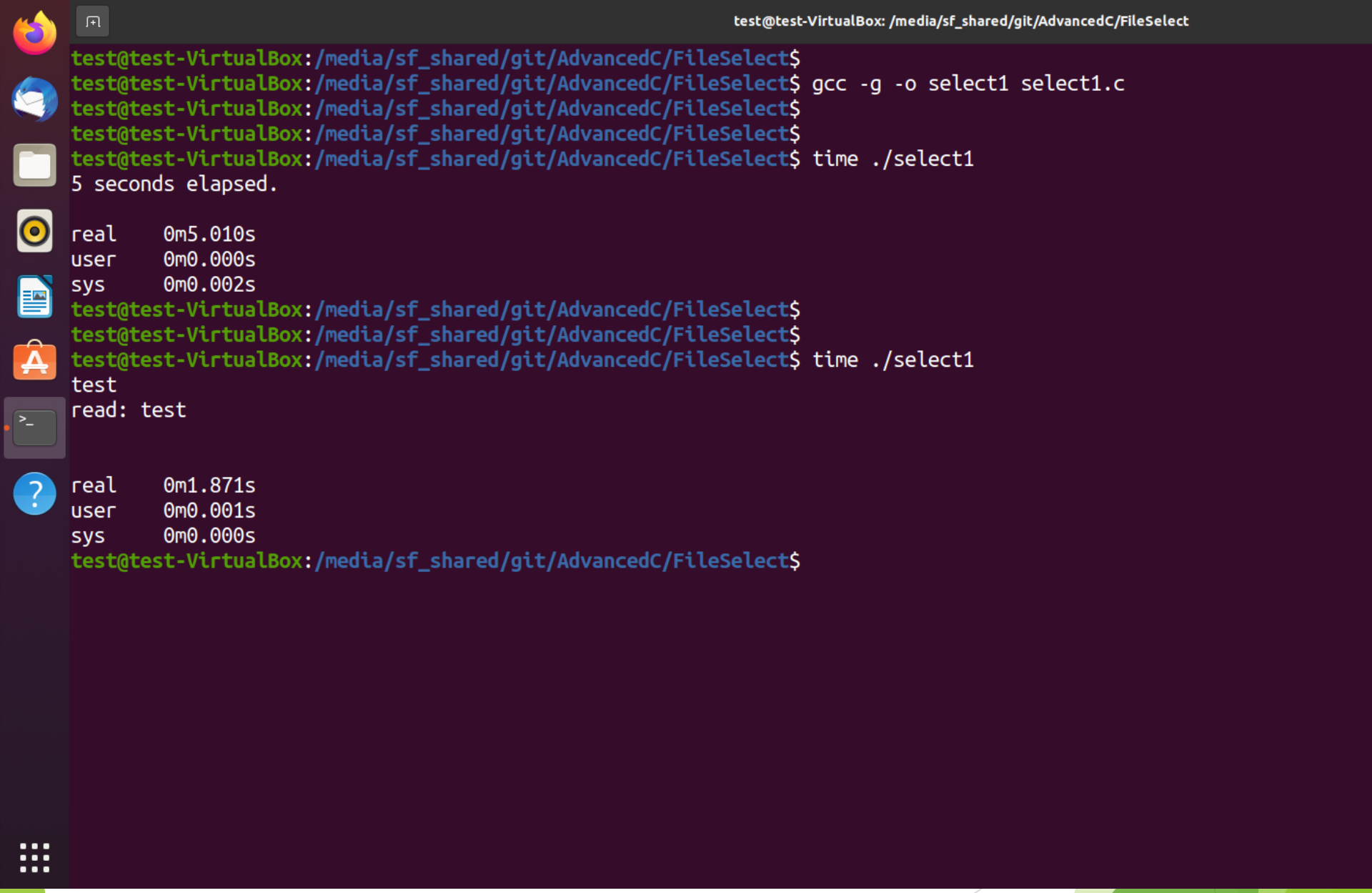
A call to `select( )` will block until the given file descriptors are ready to perform I/O, or until an optionally specified timeout has elapsed.

The `timeout` parameter is a pointer to a `timeval` structure, which is defined as follows:

```
#include <sys/time.h>

struct timeval {
        long tv_sec;          /* seconds */
        long tv_usec;         /* microseconds */
};
```


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/FileSelect
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#define TIMEOUT 5 /* select timeout in seconds */
#define BUF_LEN 1024 /* read buffer in bytes */
int main (void)
{
struct timeval tv;
fd_set readfds;
int ret;
/* Wait on stdin for input. */
FD_ZERO(&readfds);
FD_SET(STDIN_FILENO, &readfds);
/* Wait up to five seconds. */
tv.tv_sec = TIMEOUT;
tv.tv_usec = 0;
/* All right, now block! */
ret = select (STDIN_FILENO + 1,
&readfds,
NULL,
NULL,
&tv);
if (ret == -1) {
perror ("select");
return 1;
} else if (!ret) {

```
        printf ("%d seconds elapsed.\n", TIMEOUT);
        return 0;
}

/*
 * Is our file descriptor ready to read?
 * (It must be, as it was the only fd that
 * we provided and the call returned
 * nonzero, but we will humor ourselves.)
 */
if (FD_ISSET(STDIN_FILENO, &readfds)) {
        char buf[BUF_LEN+1];
        int len;

        /* guaranteed to not block */
        len = read (STDIN_FILENO, buf, BUF_LEN);
        if (len == -1) {
                perror ("read");
                return 1;
        }

        if (len) {
                buf[len] = '\0';
                printf ("read: %s\n", buf);
        }

        return 0;
}

fprintf (stderr, "This should not happen!\n");

return 1;
```


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/FileSelect
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$ gcc -g -o select1 select1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$ time ./select1
5 seconds elapsed.
real 0m5.010s
user 0m0.000s
sys 0m0.002s
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$ time ./select1
test
read: test
real 0m1.871s
user 0m0.001s
sys 0m0.000s
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileSelect$

The poll( ) system call is System V's multiplexed I/O solution. It solves several deficiencies in select( ), although select( ) is still often used (again, most likely out of habit, or in the name of portability):

```
#include <sys/poll.h>

int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

Unlike select( ), with its inefficient three bitmask-based sets of file descriptors, poll( ) employs a single array of nfds pollfd structures, pointed to by fds. The structure is defined as follows:

```
#include <sys/poll.h>

struct pollfd {
        int fd;           /* file descriptor */
        short events;     /* requested events to watch */
        short revents;    /* returned events witnessed */
};
```

Each pollfd structure specifies a single file descriptor to watch. Multiple structures may be passed, instructing poll( ) to watch multiple file descriptors. The events field of each structure is a bitmask of events to watch for on that file descriptor. The user sets this field. The revents field is a bitmask of events that were witnessed on the file descriptor. The kernel sets this field on return. All of the events requested in the events field may be returned in the revents field. Valid events are as follows:

POLLIN

There is data to read.

POLLRDNORM

There is normal data to read.

POLLRDBAND

There is priority data to read.

POLLPRI

There is urgent data to read.

POLLOUT

Writing will not block.

POLLWRNORM

Writing normal data will not block.

POLLWRBAND

Writing priority data will not block.

POLLMSG

A SIGPOLL message is available.

In addition, the following events may be returned in the revents field:

POLLER

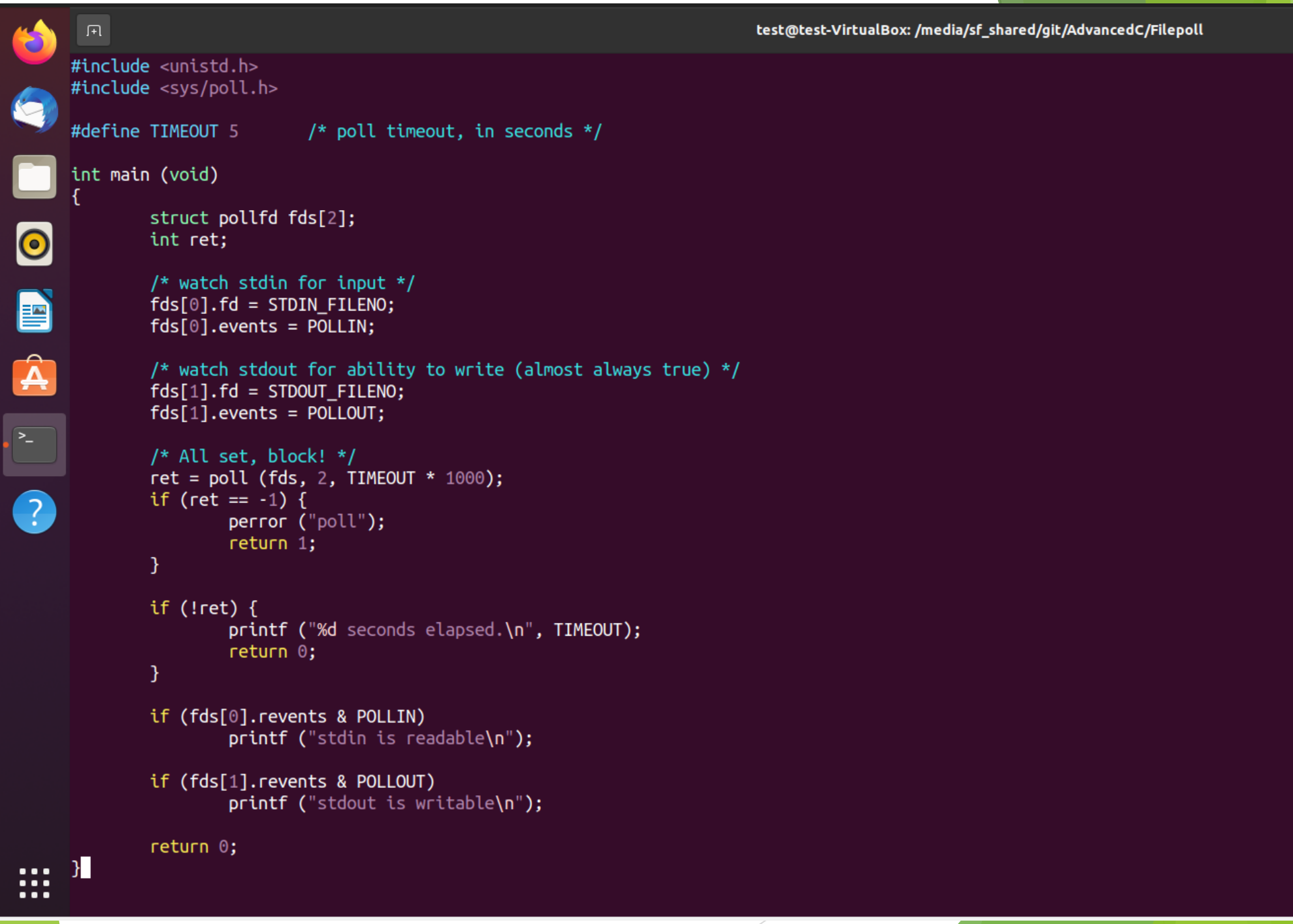
Error on the given file descriptor.

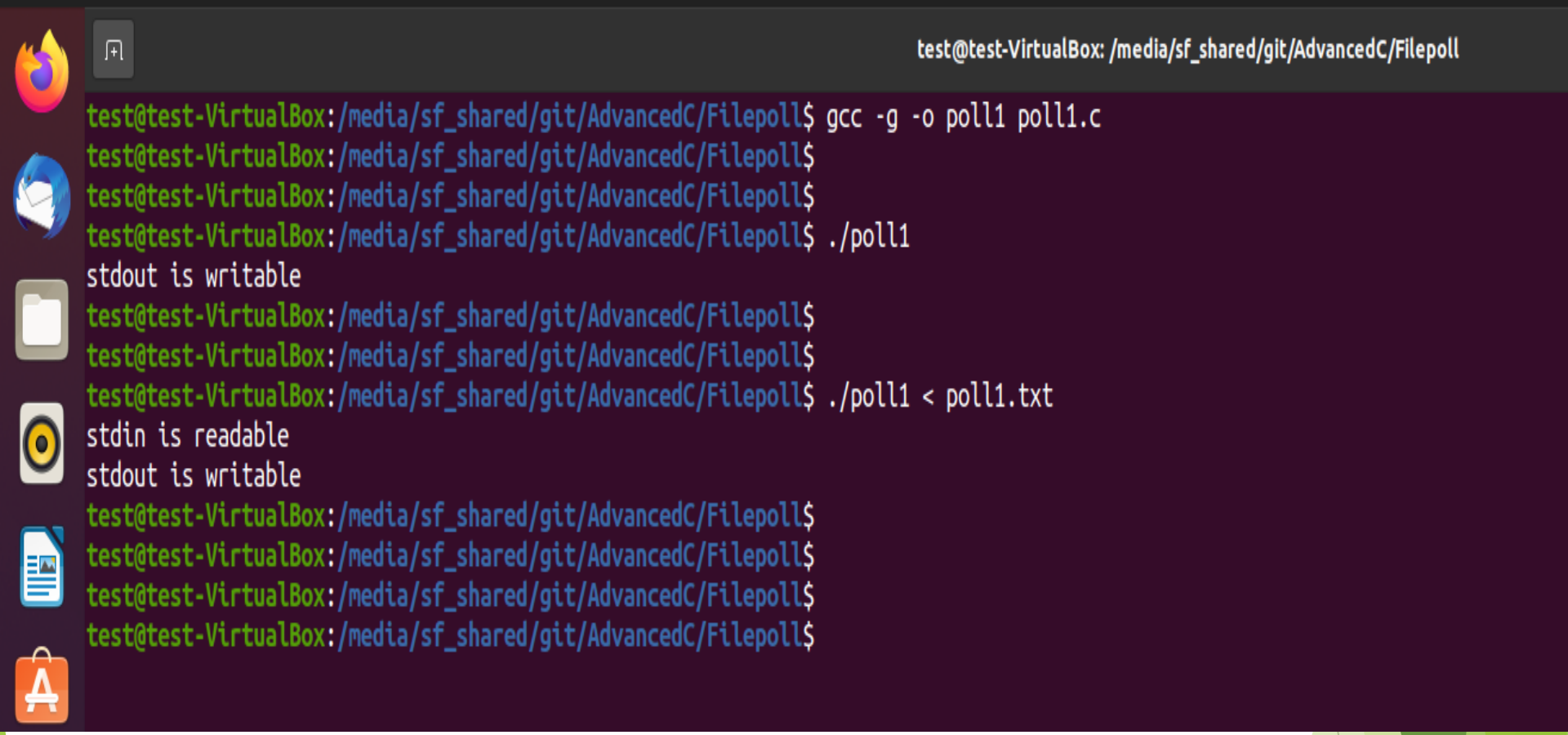
POLLHUP

Hung up event on the given file descriptor.

POLLNVAL

The given file descriptor is invalid.


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/Filepoll
#include <unistd.h>
#include <sys/poll.h>
#define TIMEOUT 5 /* poll timeout, in seconds */
int main (void)
{
struct pollfd fds[2];
int ret;
/* watch stdin for input */
fds[0].fd = STDIN_FILENO;
fds[0].events = POLLIN;
/* watch stdout for ability to write (almost always true) */
fds[1].fd = STDOUT_FILENO;
fds[1].events = POLLOUT;
/* All set, block! */
ret = poll (fds, 2, TIMEOUT * 1000);
if (ret == -1) {
perror ("poll");
return 1;
}
if (!ret) {
printf ("%d seconds elapsed.\n", TIMEOUT);
return 0;
}
if (fds[0].revents & POLLIN)
printf ("stdin is readable\n");
if (fds[1].revents & POLLOUT)
printf ("stdout is writable\n");
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/Filepoll
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$ gcc -g -o poll1 poll1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$ ./poll1
stdout is writable
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$ ./poll1 < poll1.txt
stdin is readable
stdout is writable
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Filepoll$

File Operations

- File handle : FILE *
- Open a file : fopen
- Input/Output
  - Character IO : getc, putc
  - String IO : fgets, fputs
  - Formatted IO : fscanf, fprintf
  - Raw IO : fread, fwrite
- Close a file : fclose
- Other operations :
  - fflush, fseek, freopen

Files are opened for reading or writing via `fopen( )`:

```
#include <stdio.h>

FILE * fopen (const char *path, const char *mode);
```

This function opens the file `path` according to the given modes, and associates a new stream with it.

The `fclose( )` function closes a given stream:

```
#include <stdio.h>

int fclose (FILE *stream);
```

Any buffered and not-yet-written data is first flushed. On success, `fclose( )` returns `0`. On failure, it returns `EOF` and sets `errno` appropriately.


test@test-VirtualBox: /m
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
int main()
{
FILE* fptr;
fptr=fopen("file.txt","r");
if(NULL==fptr){
perror("File open:");
exit(-1);
}
fclose(fptr);
return 0;
}

The function `fdopen( )` converts an already open file descriptor (`fd`) to a stream:

```
#include <stdio.h>

FILE * fdopen (int fd, const char *mode);
```

The possible modes are the same as for `fopen( )`, and must be compatible with the modes originally used to open the file descriptor. The modes `w` and `w+` may be specified, but they will not cause truncation. The stream is positioned at the file position associated with the file descriptor.


test@test-VirtualBox: /media/sf_shared/gi
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
int fd;
FILE* fptr;
fd=open("file.txt",O_RDONLY);
fptr=fdopen(fd,"r");
if(NULL==fptr){
perror("File open:");
exit(-1);
}
fclose(fptr);
return 0;
}

The function `fgets( )` reads a string from a given stream:

```
#include <stdio.h>

char * fgets (char *str, int size, FILE *stream);
```

This function reads up to *one less* than `size` bytes from `stream`, and stores the results in `str`. A null character (`\0`) is stored in the buffer after the bytes read in. Reading stops after an `EOF` or a newline character is reached. If a newline is read, the `\n` is stored in `str`.

For some applications, reading individual characters or lines is insufficient. Sometimes, developers want to read and write complex binary data, such as C structures. For this, the standard I/O library provides `fread( )`:

```
#include <stdio.h>

size_t fread (void *buf, size_t size, size_t nr, FILE *stream);
```

A call to `fread( )` will read up to `nr` elements of data, each of `size` bytes, from `stream` into the buffer pointed at by `buf`. The file pointer is advanced by the number of bytes read.

Individual characters and lines will not cut it when programs need to write complex data. To directly store binary data such as C variables, standard I/O provides `fwrite( )`:

```
#include <stdio.h>

size_t fwrite (void *buf,
               size_t size,
               size_t nr,
               FILE *stream);
```

A call to `fwrite( )` will write to `stream` up to `nr` elements, each `size` bytes in length, from the data pointed at by `buf`. The file pointer will be advanced by the total number of bytes written.

```
#include <stdio.h>

int fseek (FILE *stream, long offset, int whence);
```

If `whence` is set to `SEEK_SET`, the file position is set to `offset`. If `whence` is set to `SEEK_CUR`, the file position is set to the current position plus `offset`. If `whence` is set to `SEEK_END`, the file position is set to the end of the file plus `offset`.

Upon successful completion, `fseek( )` returns `0`, clears the `EOF` indicator, and undoes the effects (if any) of `ungetc( )`. On error, it returns `-1`, and `errno` is set appropriately. The most common errors are invalid stream (`EBADF`) and invalid `whence` argument (`EINVAL`).

Writing to a file using fprintf( )

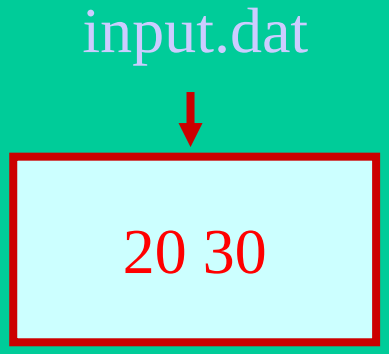
fprintf( ) works just like printf and sprintf except that its first argument is a file

```
FILE *fptr;
fptr= fopen ("file.dat","w");
/* Check it's open */
fprintf (fptr,"Hello World!\n");
```

Reading Data Using fscanf( )

- We also read data from a file using fscanf( ).

```
FILE *fptr;
fptr= fopen (“input.dat”,“r”);
/* Check it's open */
if (fptr==NULL)
  {
   printf(“Error in opening file \n”);
   }
fscanf(fptr,“%d%d”,&x,&y);
```



x=20
y=30

Unlike `lseek( )`, `fseek( )` does not return the updated position. A separate interface is provided for this purpose. The `ftell( )` function returns the current stream position of `stream`:

```
#include <stdio.h>

long ftell (FILE *stream);
```

```
#include <stdio.h>

void rewind (FILE *stream);
```

This invocation:

```
rewind (stream);
```

resets the position back to the start of the stream. It is equivalent to:

```
fseek (stream, 0, SEEK_SET);
```

except that it also clears the error indicator.

File operation Handson

Anandkumar

Logging Events

- A program may log events and conditions during it's run time.

- In Linux, logging can be done in program itself, or by assistance of Linux facilities.

- Using *SYSLOG*, a program can log events in different levels of priority.

- To log events with syslog, the "syslog" service must be started in system.

Logging Events

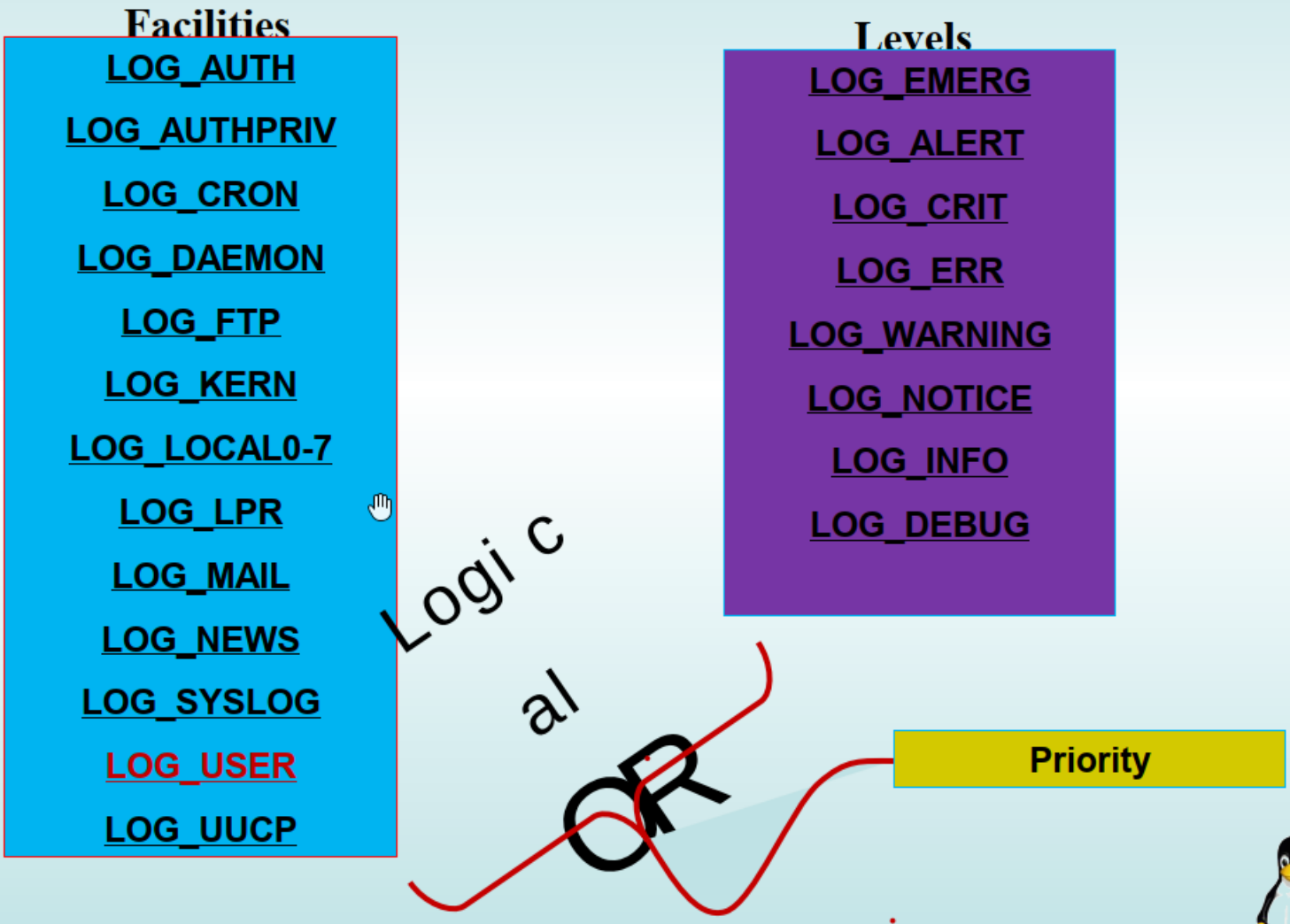
- You can use the “syslog” function in your program in order to log the events:

syslog (int priority, const char * format, …)

Priority = facility || level

Format and format strings are the same as we use in printf

Logging Events



Logging Events

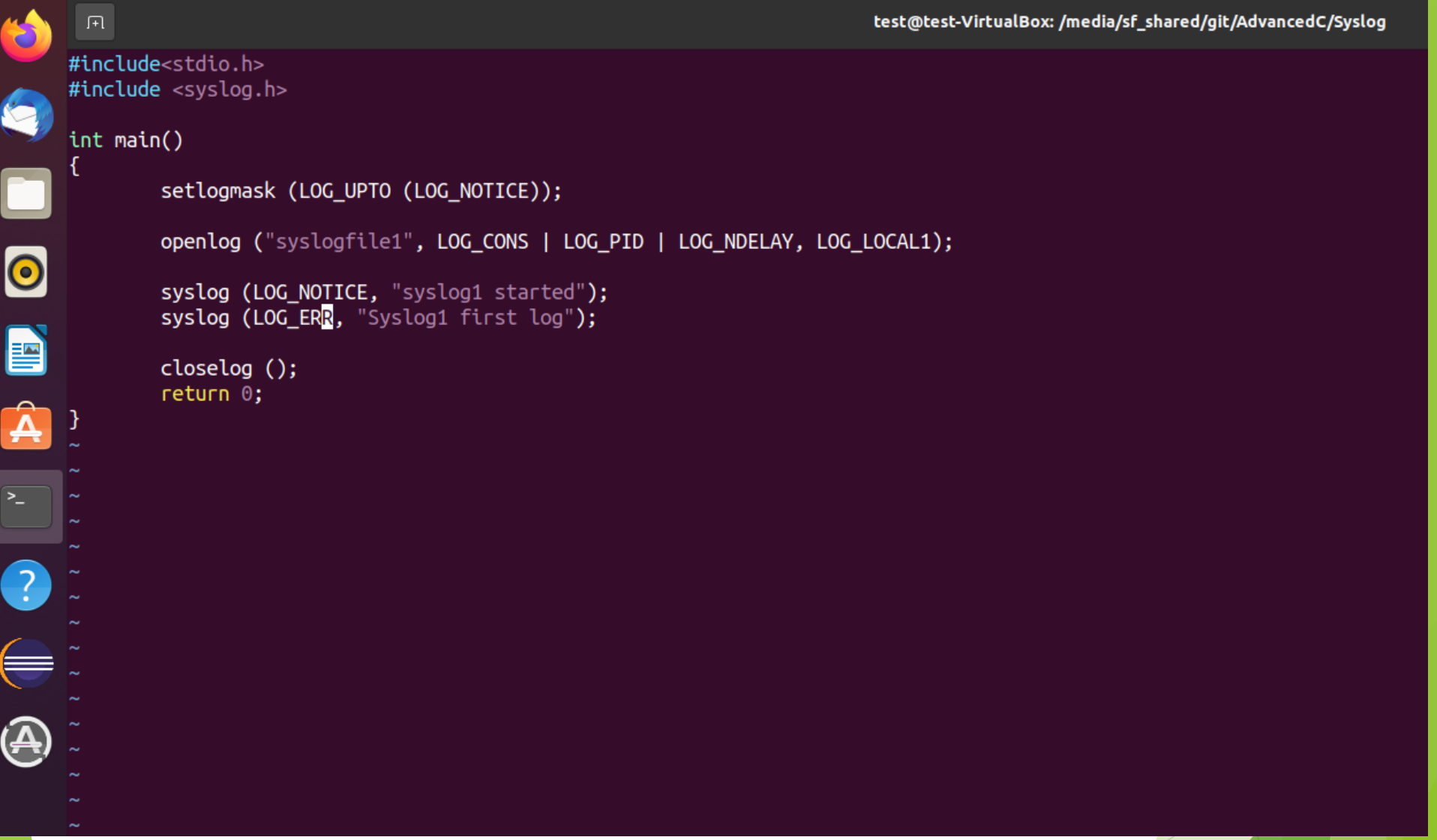
- The “syslog” function will generate a log message.

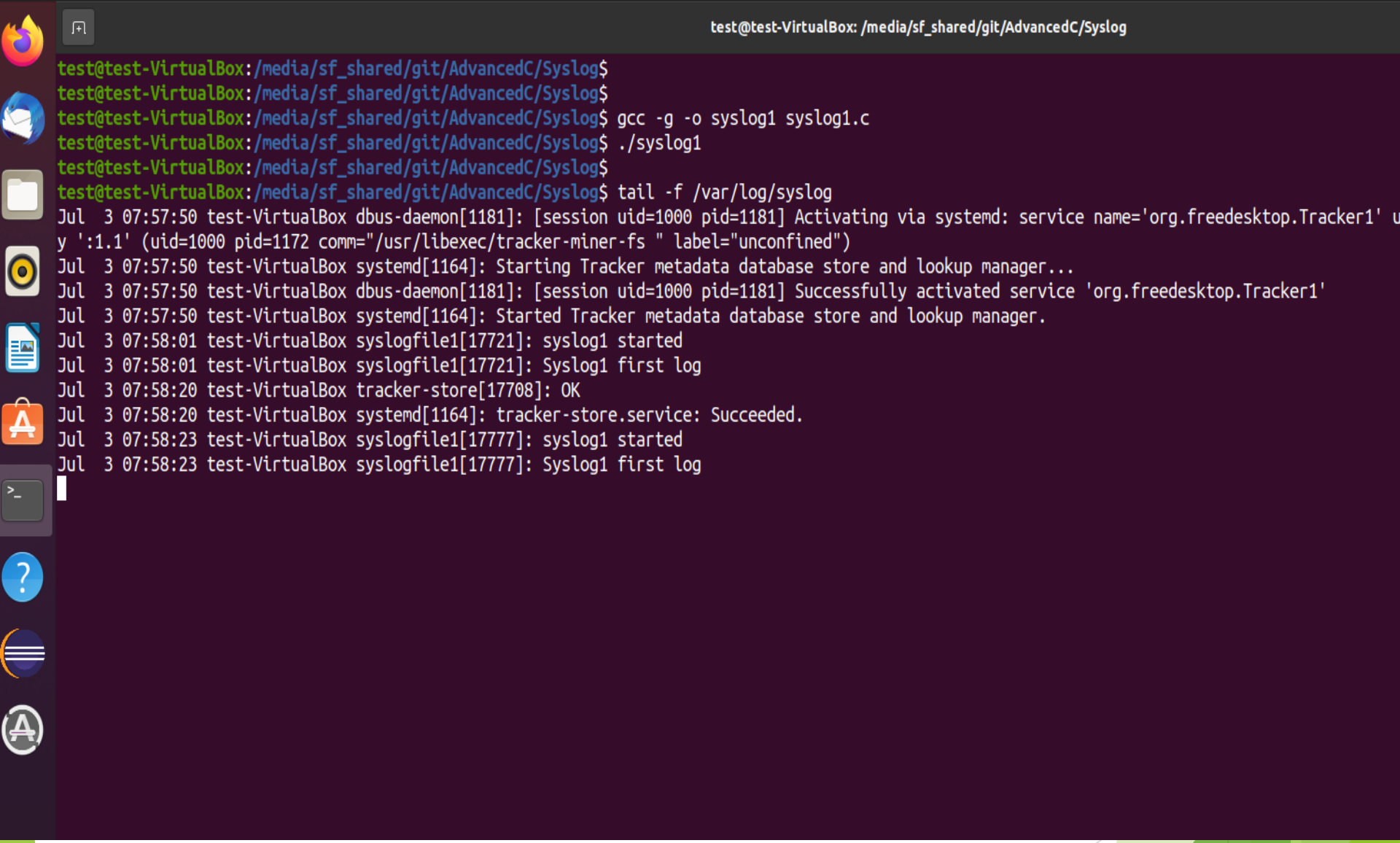
- The log message will be distributed by “syslogd”.

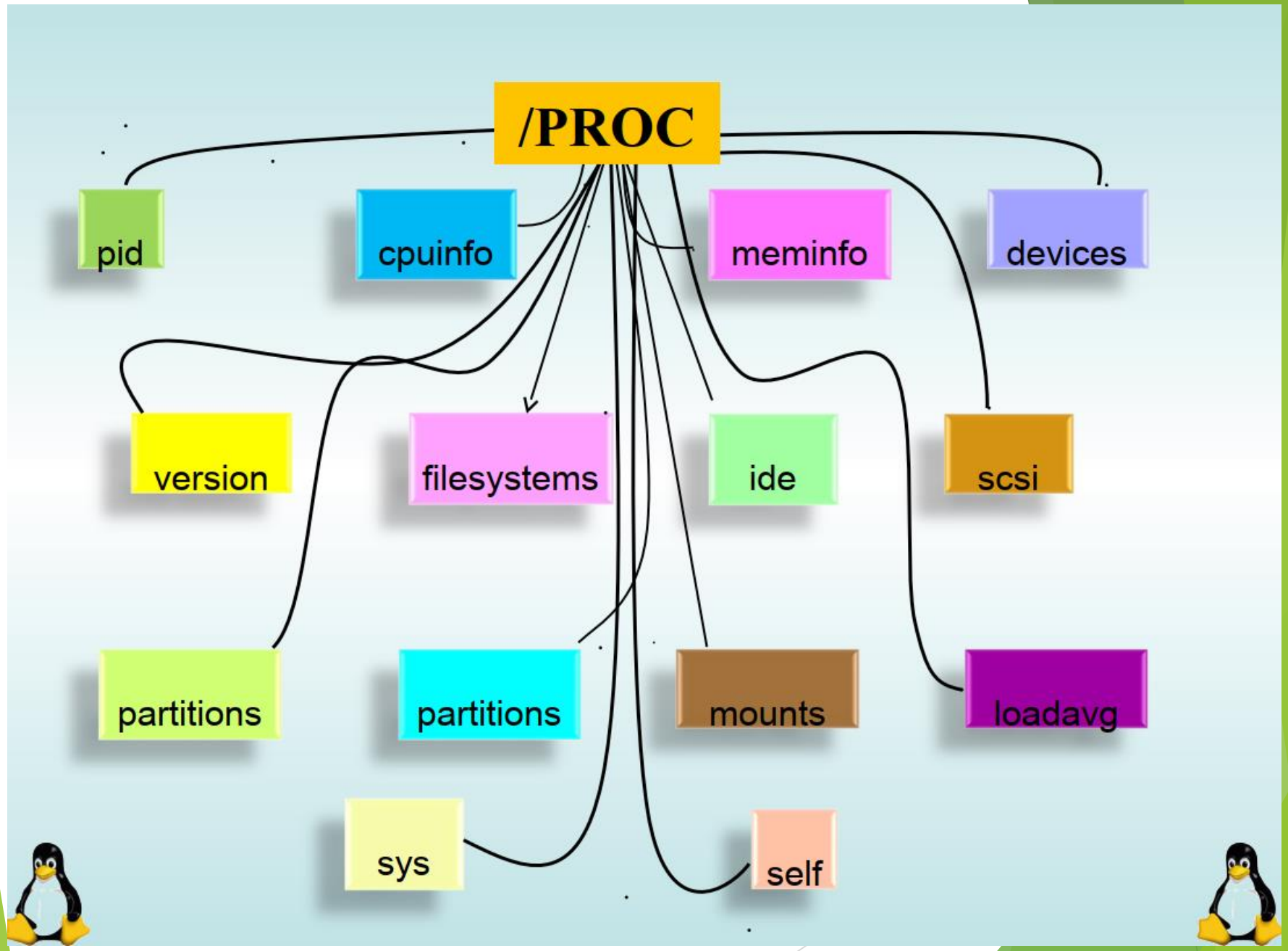
- The “syslogd” (syslog daemon) should be well configured to work correctly.

- Configuration file for syslogd is “/etc/syslog.conf.

- Syslog daemon will write the logs to appropriate files and sends them to suitable devices according to configuration file.


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/Syslog
#include<stdio.h>
#include <syslog.h>
int main()
{
setlogmask (LOG_UPTO (LOG_NOTICE));
openlog ("syslogfile1", LOG_CONS | LOG_PID | LOG_NDELAY, LOG_LOCAL1);
syslog (LOG_NOTICE, "syslog1 started");
syslog (LOG_ERR, "Syslog1 first log");
closelog ();
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/Syslog
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Syslog$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Syslog$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Syslog$ gcc -g -o syslog1 syslog1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Syslog$ ./syslog1
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Syslog$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/Syslog$ tail -f /var/log/syslog
Jul 3 07:57:50 test-VirtualBox dbus-daemon[1181]: [session uid=1000 pid=1181] Activating via systemd: service name='org.freedesktop.Tracker1' u
y ':1.1' (uid=1000 pid=1172 comm="/usr/libexec/tracker-miner-fs " label="unconfined")
Jul 3 07:57:50 test-VirtualBox systemd[1164]: Starting Tracker metadata database store and lookup manager...
Jul 3 07:57:50 test-VirtualBox dbus-daemon[1181]: [session uid=1000 pid=1181] Successfully activated service 'org.freedesktop.Tracker1'
Jul 3 07:57:50 test-VirtualBox systemd[1164]: Started Tracker metadata database store and lookup manager.
Jul 3 07:58:01 test-VirtualBox syslogfile1[17721]: syslog1 started
Jul 3 07:58:01 test-VirtualBox syslogfile1[17721]: Syslog1 first log
Jul 3 07:58:20 test-VirtualBox tracker-store[17708]: OK
Jul 3 07:58:20 test-VirtualBox systemd[1164]: tracker-store.service: Succeeded.
Jul 3 07:58:23 test-VirtualBox syslogfile1[17777]: syslog1 started
Jul 3 07:58:23 test-VirtualBox syslogfile1[17777]: Syslog1 first log


/PROC
pid
cpuinfo
meminfo
devices
version
filesystems
ide
scsi
partitions
partitions
mounts
loadavg
sys
self

/PROC

- Is a pseudo filesystem which contains the processes information.

- Is not associated with a hardware device (like disk devices).

- Is a window to the running kernel.

- Contents of the files in this directory are not fixed blocks and are generated bye the Linux kernel when you read them.

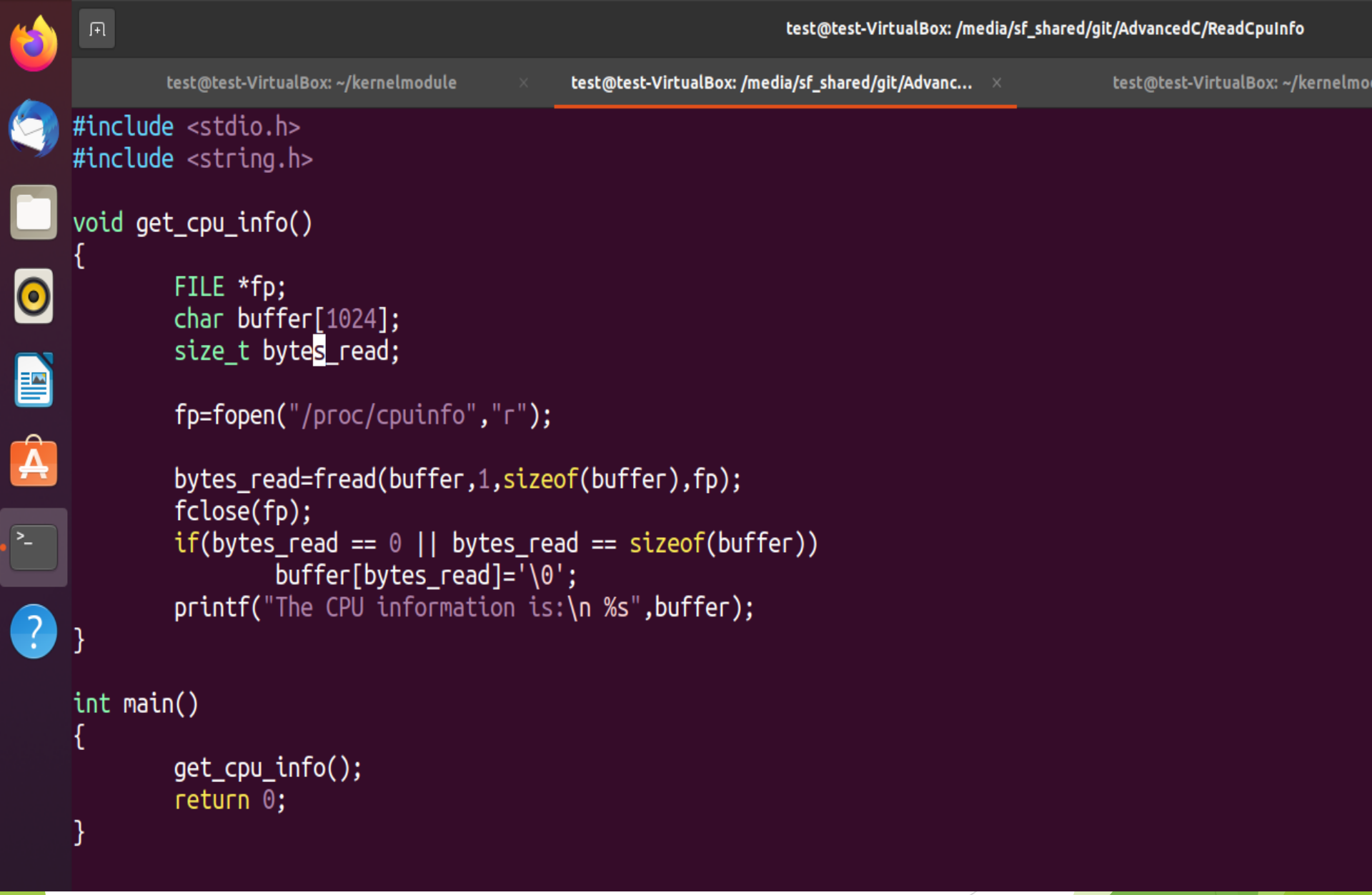
- Some files in /proc allow kernel variables to be changed.

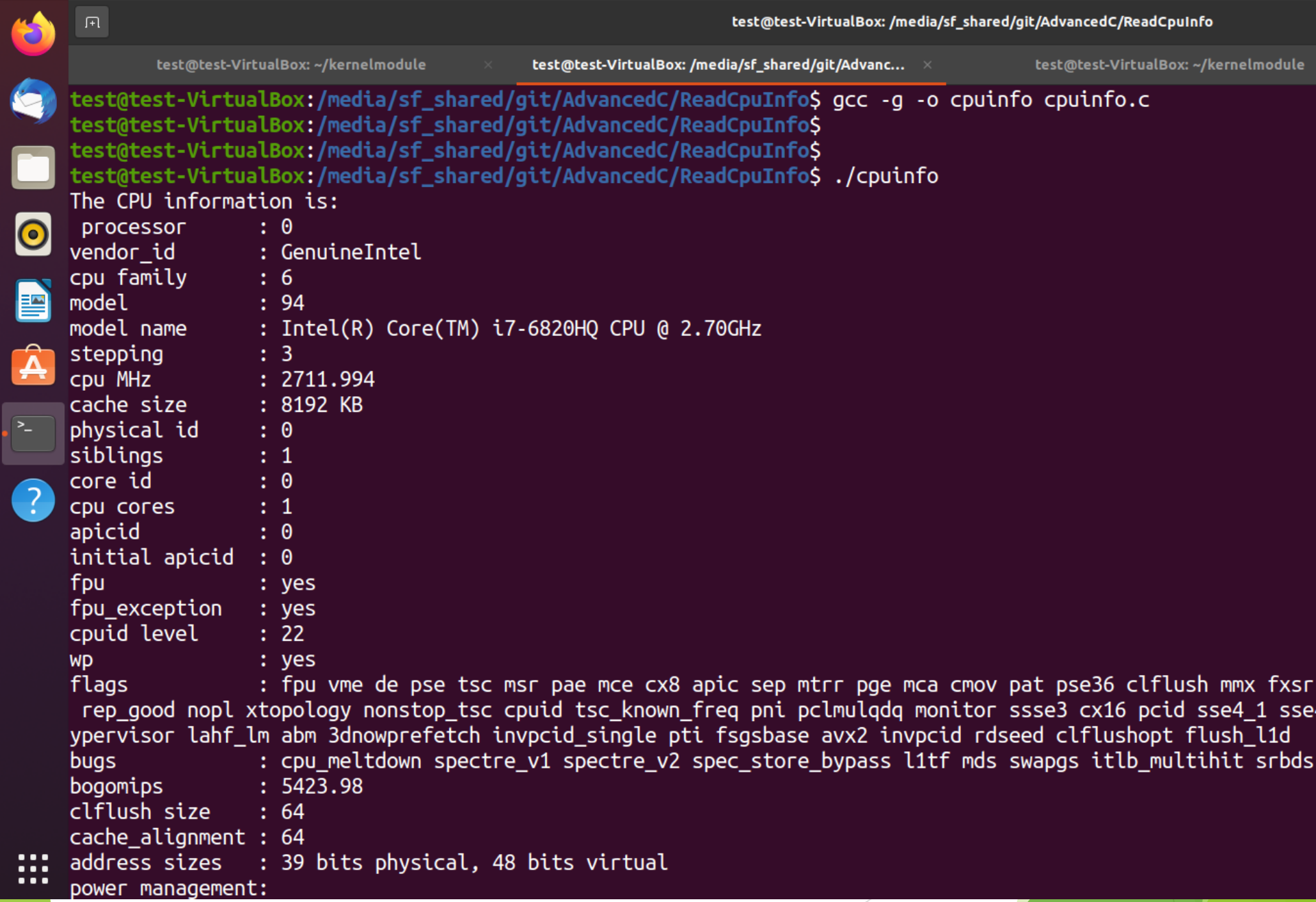
/PROC

- You can get the information you want by reading the contents of /proc files in your program.

- Some of these files are:

/proc/cpuinfo	Information about CPU (s)
/proc/version	Version of Linux kernel (uname)
/proc/meminfo	Information about memory usage
/proc/filesystems	What filesystem types are loaded in kernel right now
/proc/mounts	What filesystems are mounted


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ReadCpuInfo
test@test-VirtualBox: ~/kernelmodule
test@test-VirtualBox: /media/sf_shared/git/Advanc...
test@test-VirtualBox: ~/kernelmo
#include <stdio.h>
#include <string.h>
void get_cpu_info()
{
FILE *fp;
char buffer[1024];
size_t bytes_read;
fp=fopen("/proc/cpuinfo","r");
bytes_read=fread(buffer,1,sizeof(buffer),fp);
fclose(fp);
if(bytes_read == 0 || bytes_read == sizeof(buffer))
buffer[bytes_read]='\0';
printf("The CPU information is:\n %s",buffer);
}
int main()
{
get_cpu_info();
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ReadCpuInfo
test@test-VirtualBox: ~/kernelmodule
test@test-VirtualBox: /media/sf_shared/git/Advanc...
test@test-VirtualBox: ~/kernelmodule
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ReadCpuInfo$ gcc -g -o cpuinfo cpuinfo.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ReadCpuInfo$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ReadCpuInfo$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ReadCpuInfo$ ./cpuinfo
The CPU information is:
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 94
model name : Intel(R) Core(TM) i7-6820HQ CPU @ 2.70GHz
stepping : 3
cpu MHz : 2711.994
cache size : 8192 KB
physical id : 0
siblings : 1
core id : 0
cpu cores : 1
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 22
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr
rep_good nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq monitor ssse3 cx16 pcid sse4_1 sse
ypervisor lahf_lm abm 3dnowprefetch invpcid_single pti fsgsbase avx2 invpcid rdseed clflushopt flush_l1d
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit srbds
bogomips : 5423.98
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:

/PROC

- each process has a directory specified to it in /proc (pid).

- in each pid directory, there are some information about the process.

- the “self” directory, points to the running process itself.

- in each process directory there are some files and subdirectories indicating some information about that process.

- cmdline, cwd, environ, exe, fd, maps, mem, root , … (see man 5 proc)

Creating Processes in Linux

- The simple way: using *system* function.

- The flexible, secure, complex way: using *fork* and *exec*

- By using *system* you can create a subprocess running the standard Bourne shell (/bin/sh) and execute a command in it.

- By using *fork* function you can create a child process which is an exact copy of it's parent.

- By using *exec* family of functions, you can replace the current process image with a new one.

DOS and Windows API use *spawn* family instead of *fork* & *exec*

Creating Processes in Linux

- The *system* function, uses a shell to invoke the desired program.

- It has the same features, limitations, and security flaws of the system's shell.

```
int system (const char * command )
```

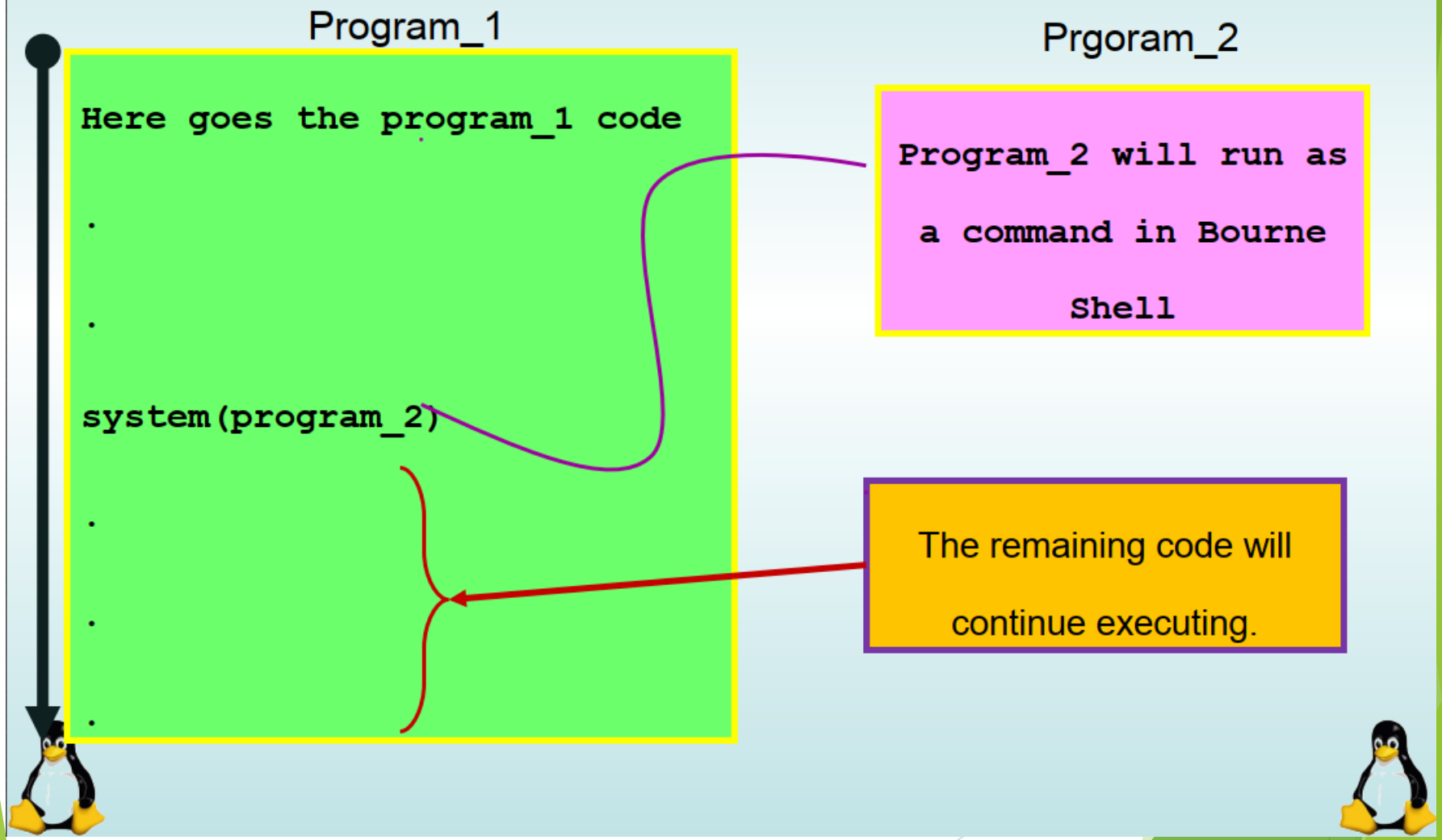
system will return the exit status of the command (see *wait*).

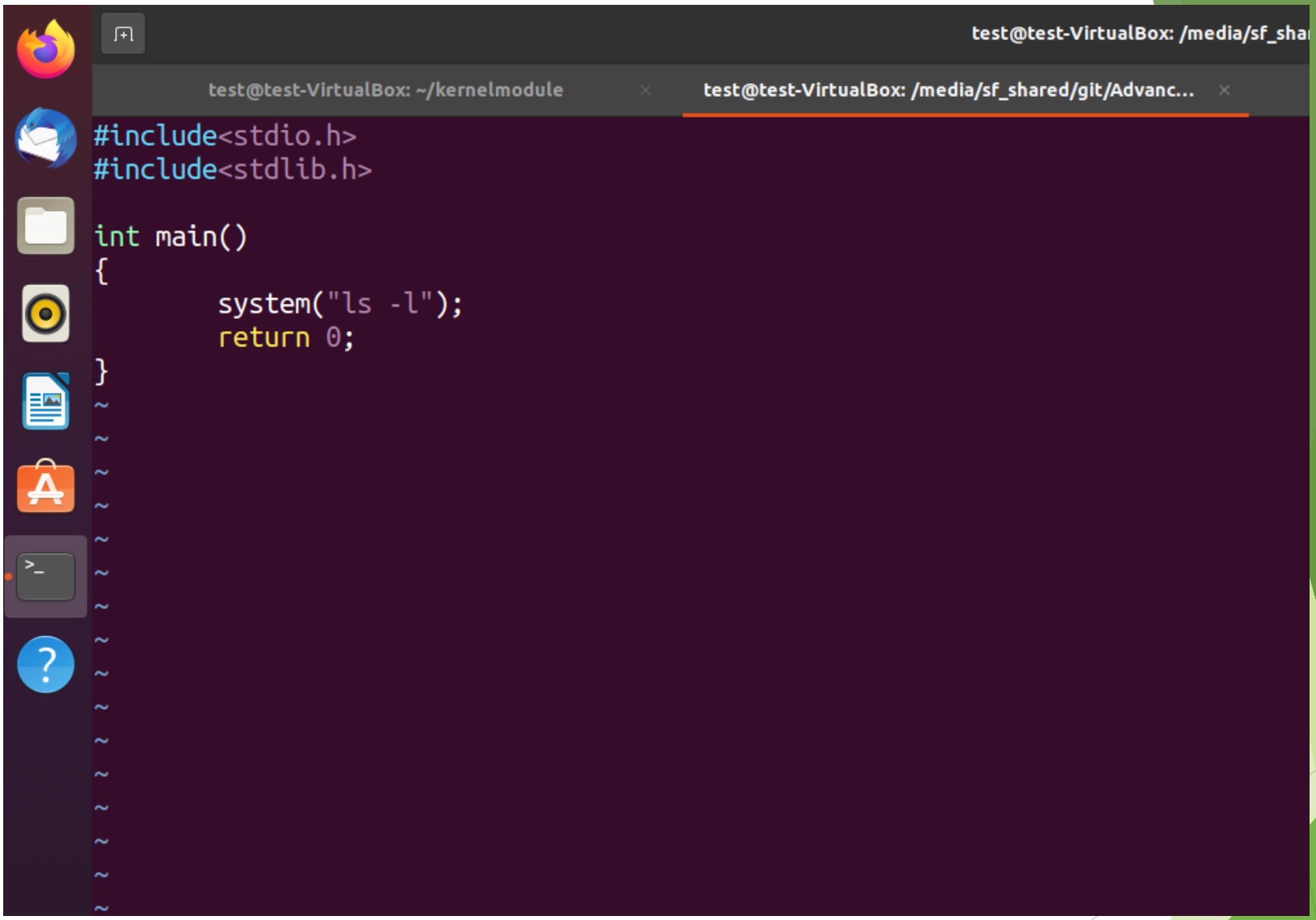
127: shell can not be run

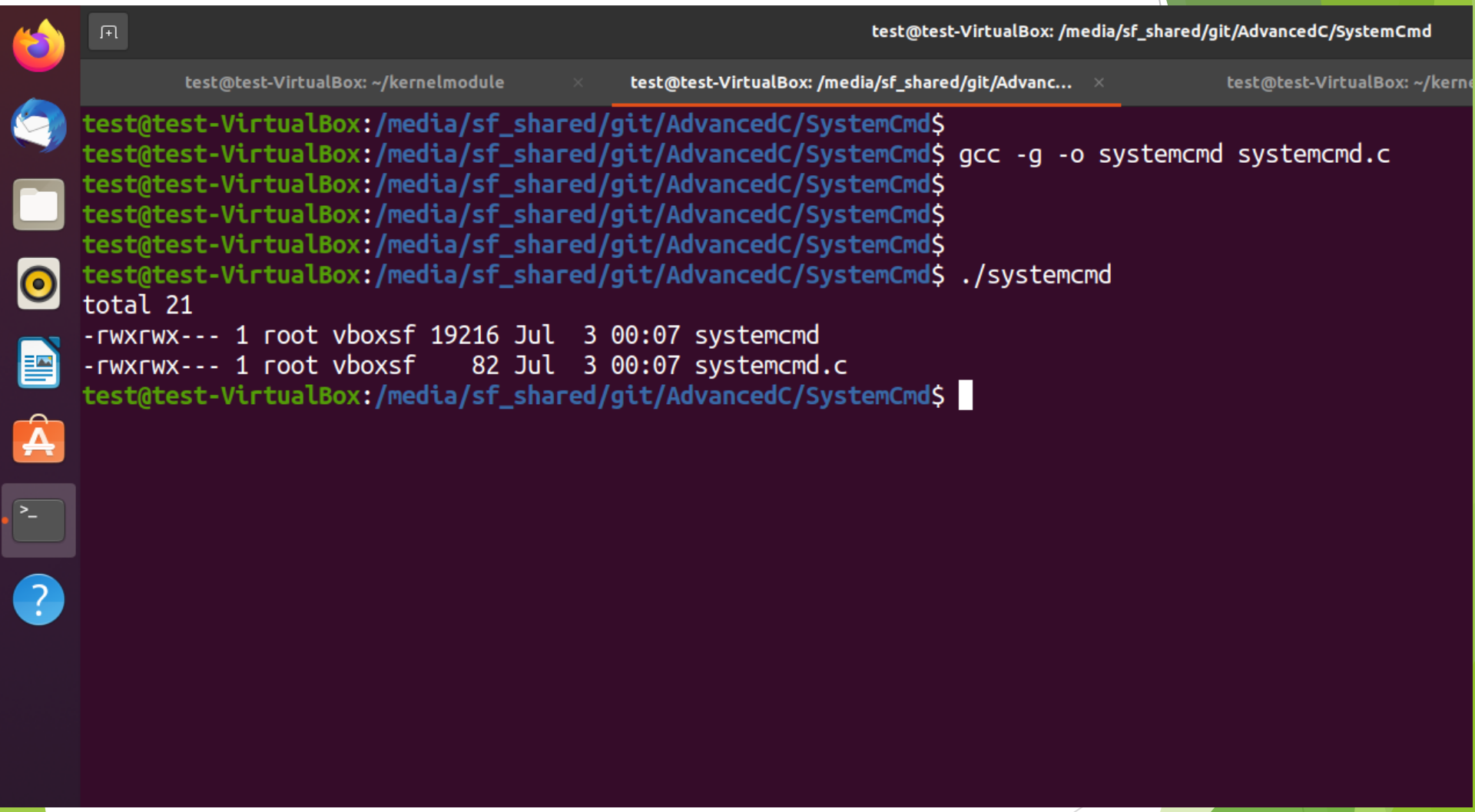
-1: any other errors

system will call this command by calling *"/bin/sh -c command"*

Creating Processes in Linux




test@test-VirtualBox: /media/sf_sha
test@test-VirtualBox: ~/kernelmodule
test@test-VirtualBox: /media/sf_shared/git/Advanc...
#include<stdio.h>
#include<stdlib.h>
int main()
{
system("ls -l");
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/SystemCmd
test@test-VirtualBox: ~/kernelmodule
test@test-VirtualBox: /media/sf_shared/git/Advanc...
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$ gcc -g -o systemcmd systemcmd.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$ ./systemcmd
total 21
-rwxrwx--- 1 root vboxsf 19216 Jul 3 00:07 systemcmd
-rwxrwx--- 1 root vboxsf 82 Jul 3 00:07 systemcmd.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/SystemCmd$

Creating Processes in Linux

- The *fork* function creates child process which only differ in its PID with his parent.

- Return value in parent process is PID of child and in child is 0.

```
pid_t fork ( void )
```

fork will return a PID (PID of child or zero) on success and -1 on failure.

fork does not need any additional arguments. It just creates the same process as the parent.

fork()

- A process calling `fork()` spawns a child process.
- The child is almost an identical *clone* of the parent:
  - Program Text (segment .text)
  - Stack (ss)
  - PCB (eg. registers)
  - Data (segment .data)

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

pid_t fork(void);
```

Creating a Process – fork()

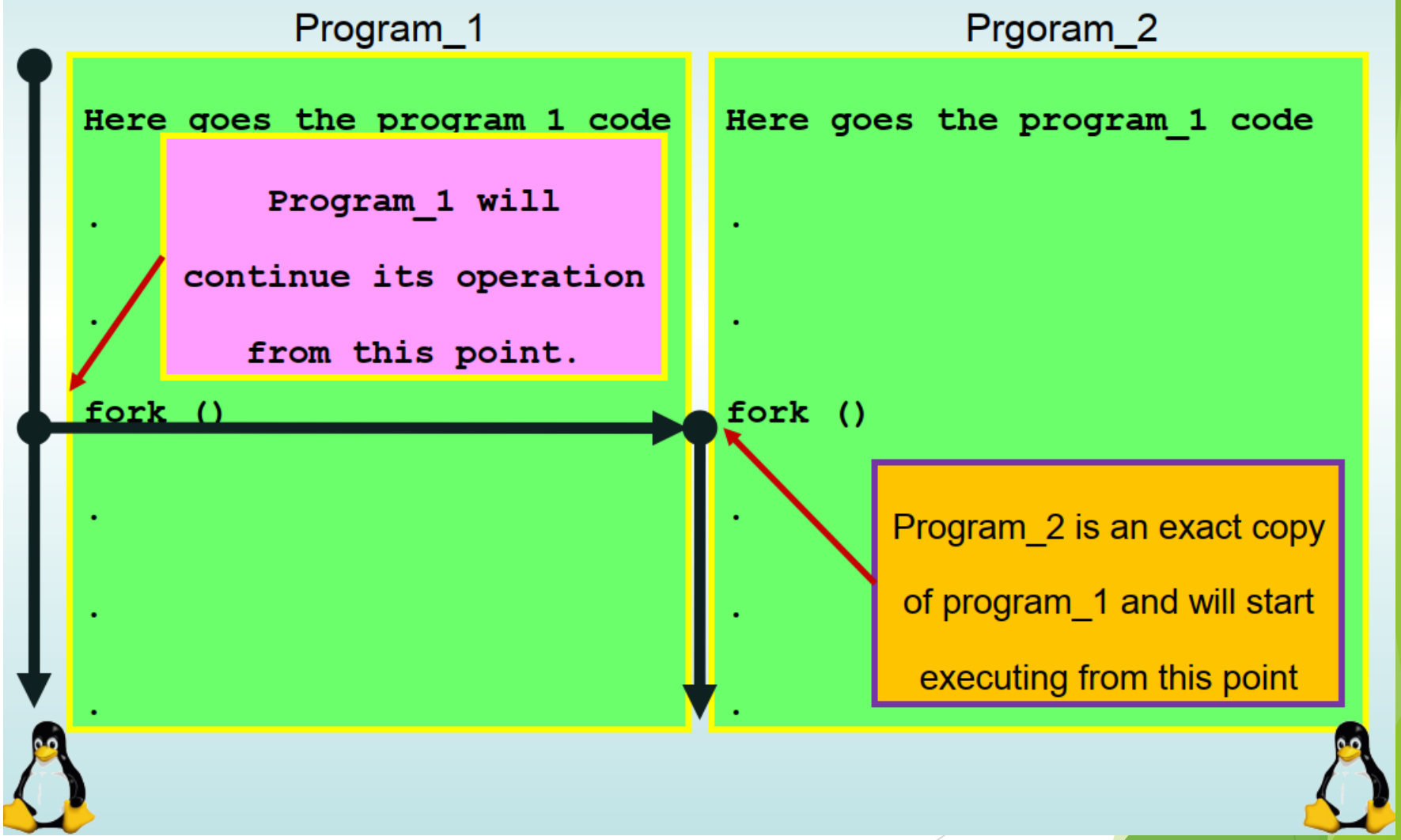
pid_t fork() creates a duplicate of the calling process:

- Both processes continue with *return from fork()*
- Child gets exact copy of code, stack, file descriptors, heap, globals, and program counter

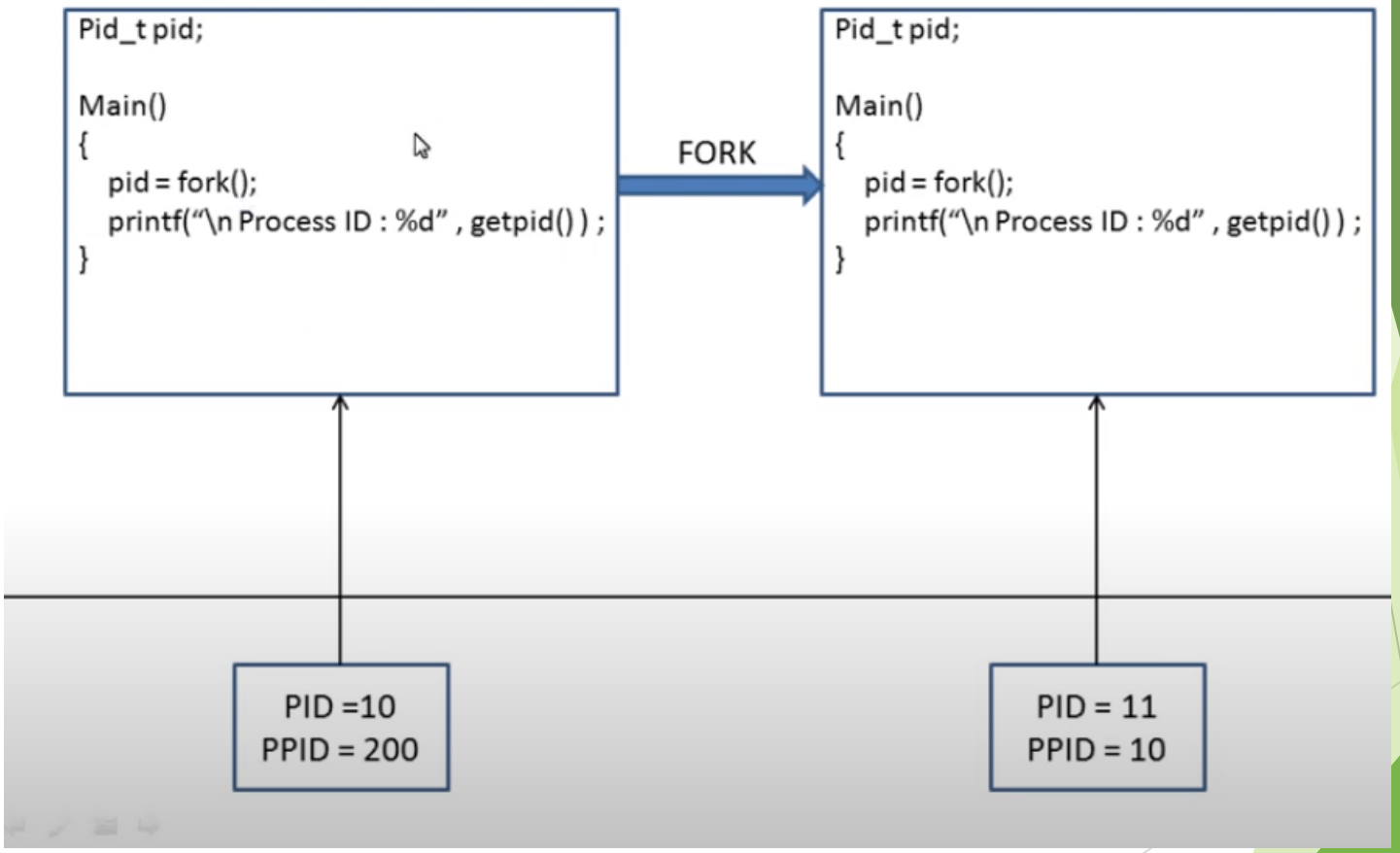
fork() returns

- -1 if fork fails
- 0 in child process
- child's PID in parent process

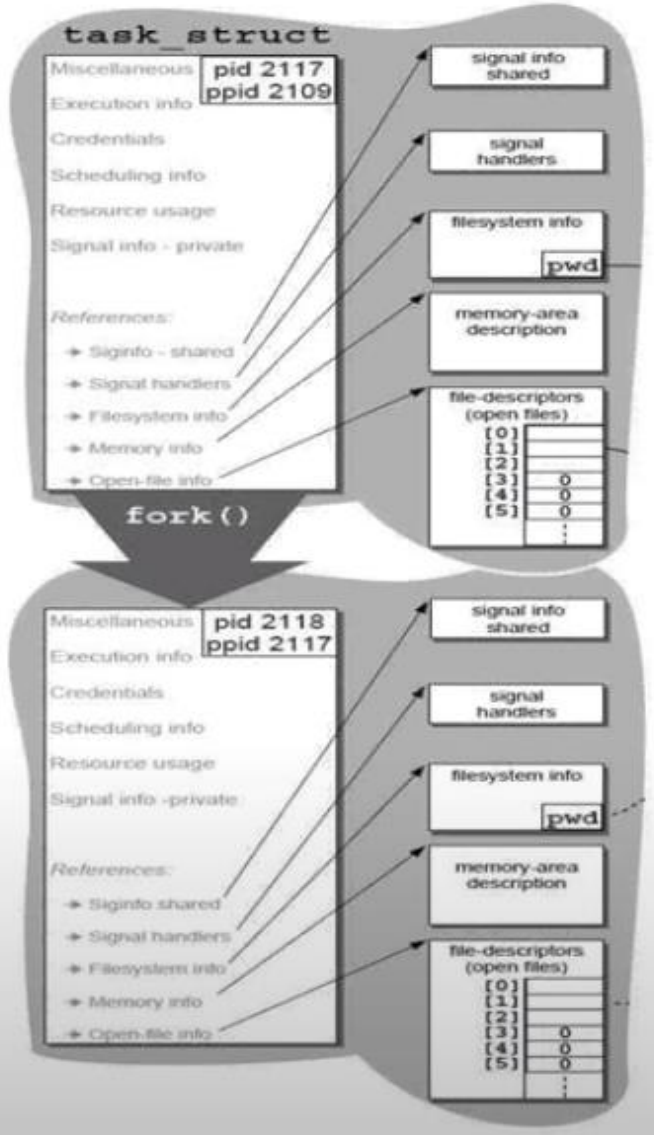
Creating Processes in Linux



Process creation with fork()

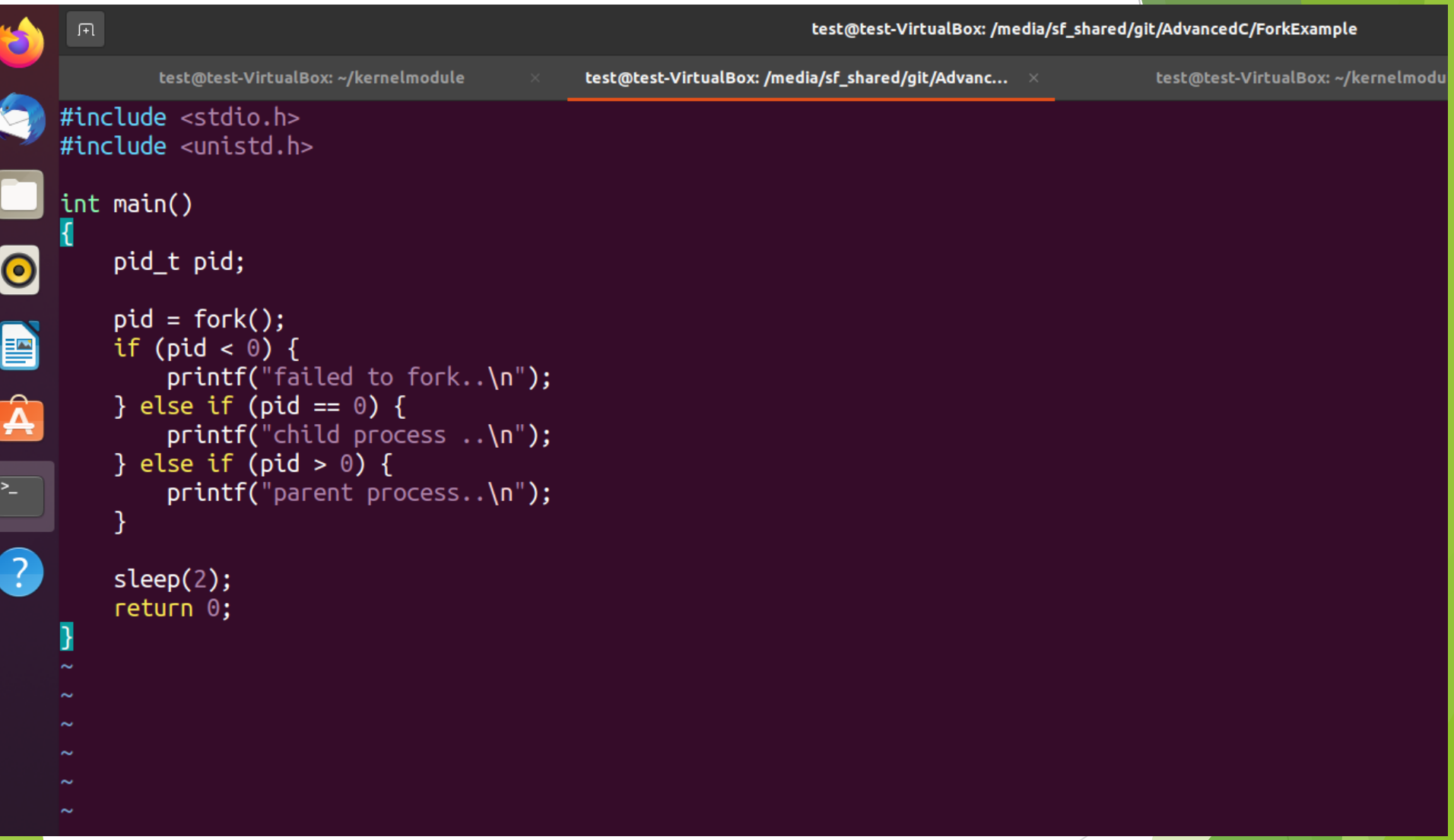


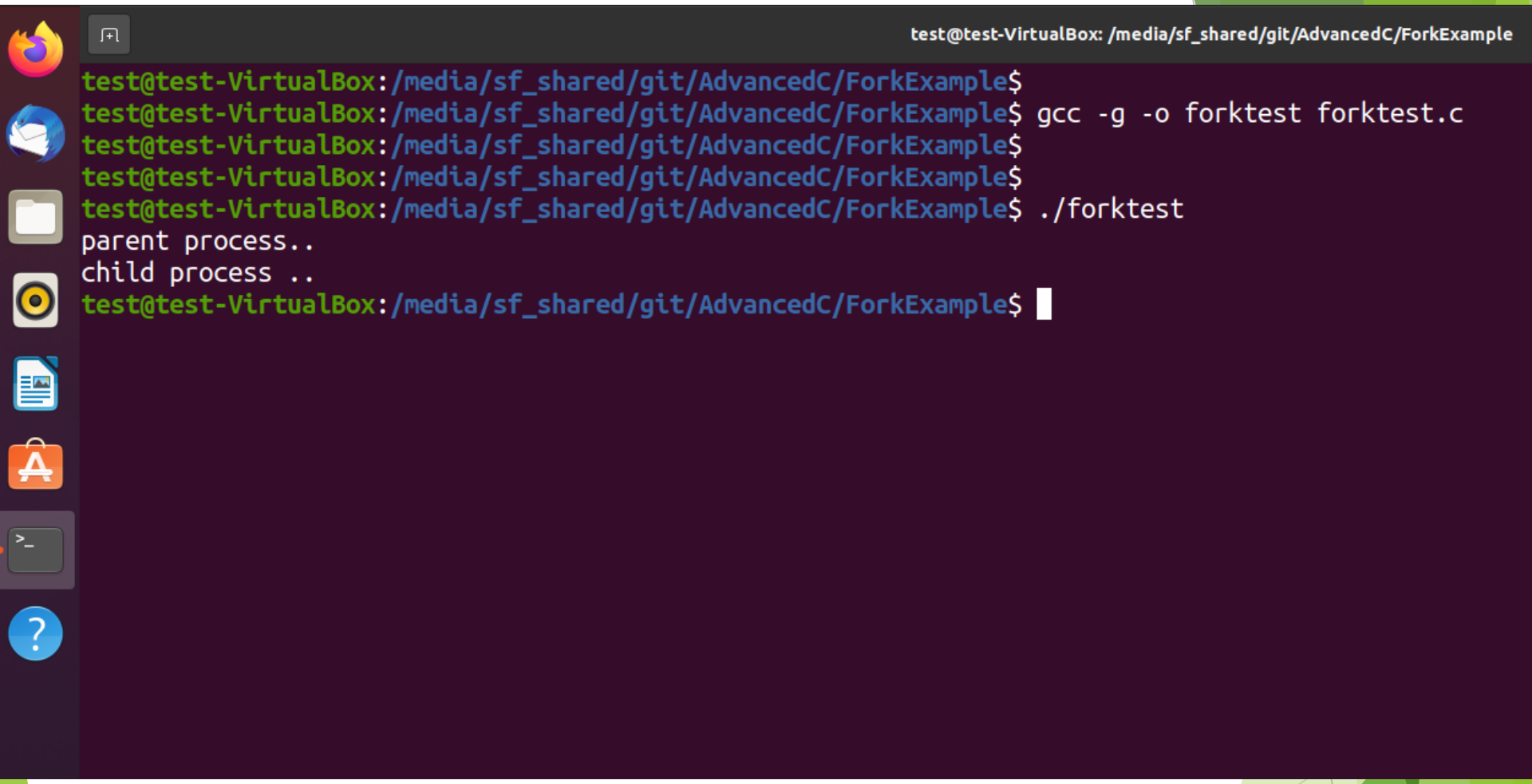
Process creation with fork()



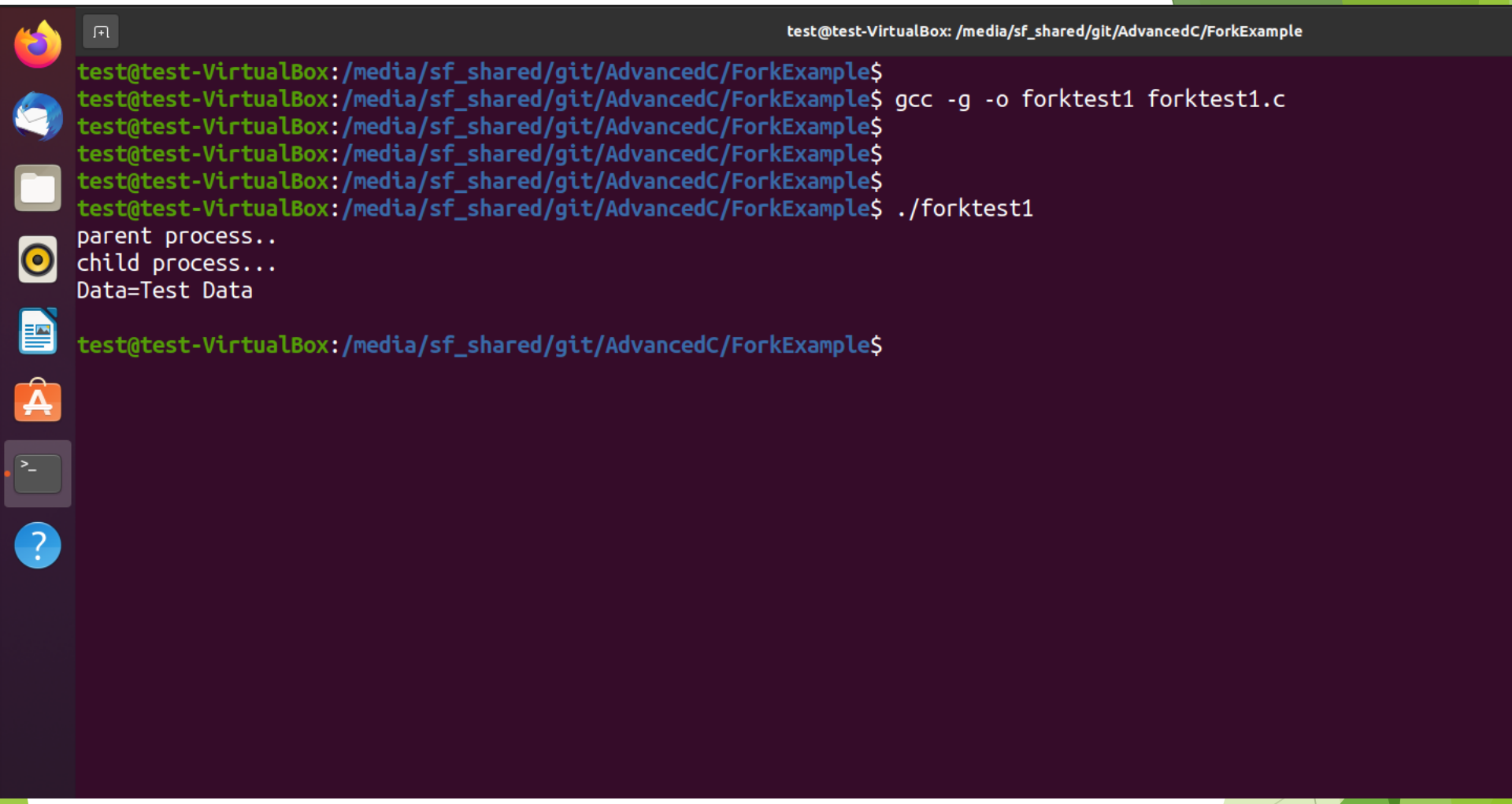
fork → sys_fork → do_fork ($SRC_HOME/kernel/fork.c)

- ✓ fail if #processes for this user exceeded
- ✓ increment #processes for this user
- ✓ fail if #processes in system exceeds maximum
  nr_threads >= max_threads
- ✓ Allocate new ***task_struct*** instance
- ✓ determine pid for child
- ✓ clear pending signals for child
- ✓ Copy the satellite structures for the child process.
  - copy ***files_struct***
  - copy ***fs_struct***
  - copy ***signal_struct***
  - copy ***sighand_struct***
  - copy ***mm_struct***
- ✓ copy ***cpu-registers*** from parent to child; fill **eip** of child with other continuation address
- ✓ set child's state to '***running***', add to runqueue and force process-switch to child
- ✓ Return PID of child to parent


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkExample
test@test-VirtualBox: ~/kernelmodule
test@test-VirtualBox: /media/sf_shared/git/Advanc...
test@test-VirtualBox: ~/kernelmodu
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
pid = fork();
if (pid < 0) {
printf("failed to fork..\n");
} else if (pid == 0) {
printf("child process ..\n");
} else if (pid > 0) {
printf("parent process..\n");
}
sleep(2);
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkExample
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$ gcc -g -o forktest forktest.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$ ./forktest
parent process..
child process ..
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkExample
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
pid_t pid;
int fd=0;
char buf[256];
fd=open("./testfile",O_RDONLY);
pid = fork();
if (pid < 0) {
printf("failed to fork..\n");
} else if (pid == 0) {
printf("child process...\n");
read(fd,buf,sizeof(buf));
printf("Data=%s\n",buf);
} else if (pid > 0) {
printf("parent process..\n");
}
sleep(2);
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkExample
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$ gcc -g -o forktest1 forktest1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$ ./forktest1
parent process..
child process...
Data=Test Data
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkExample$

Process Identification

UNIX identifies processes via a unique Process ID

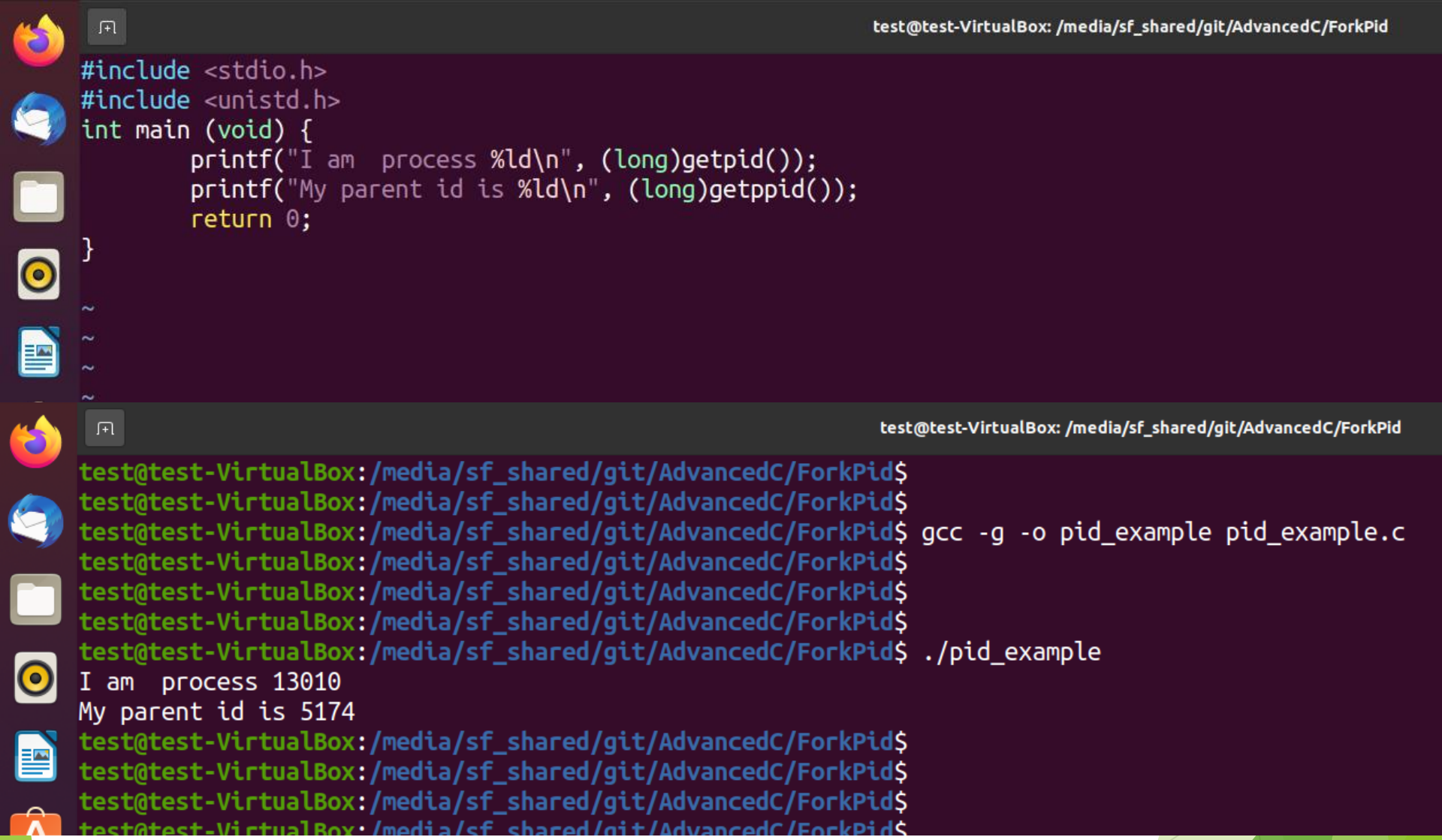
- Each process also knows its parent process ID since each process is created from a parent process.
- Root process is the 'init' process

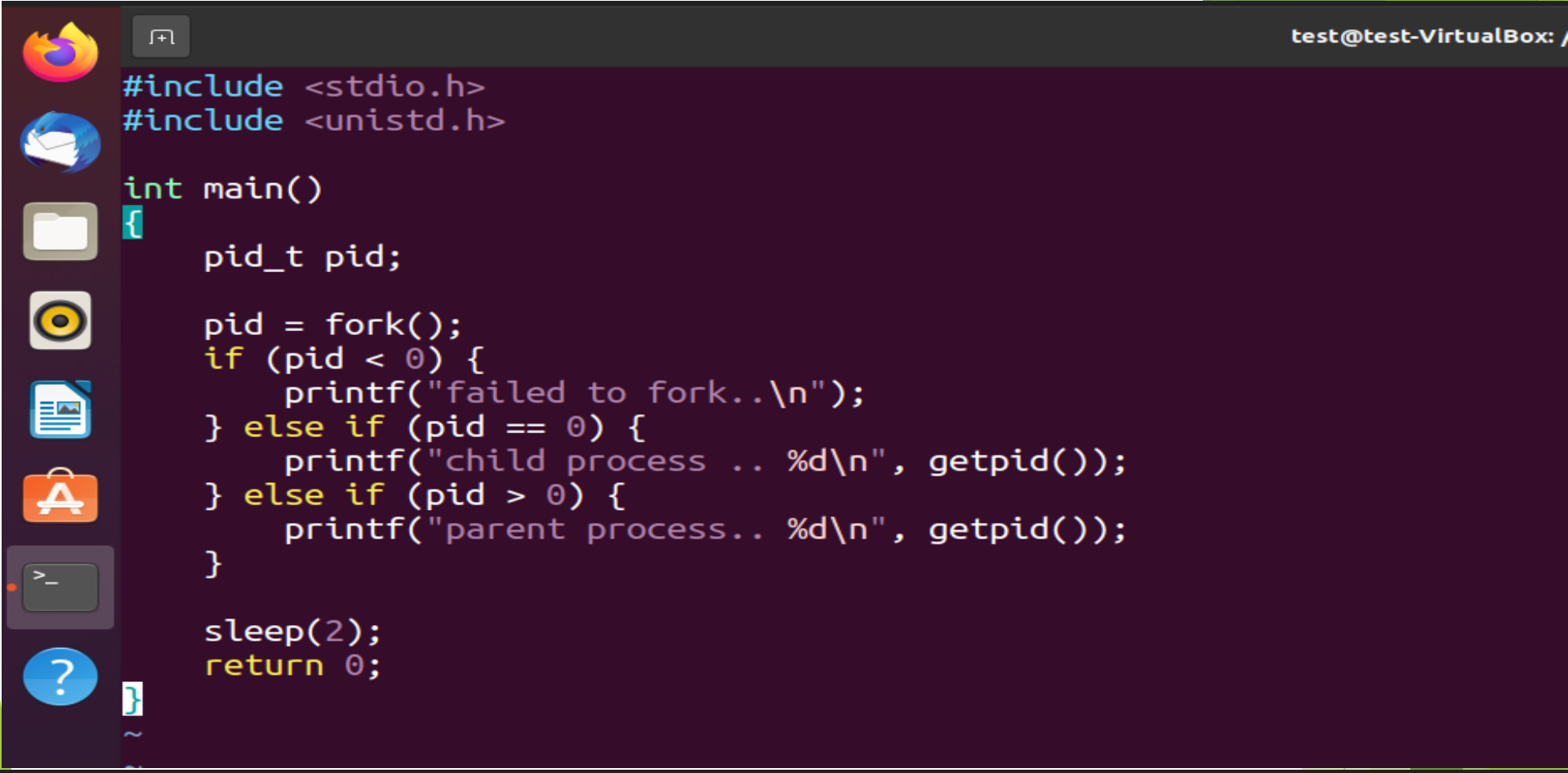
'*getpid*' and '*getppid*' – functions to return process ID (PID) and parent process ID (PPID)

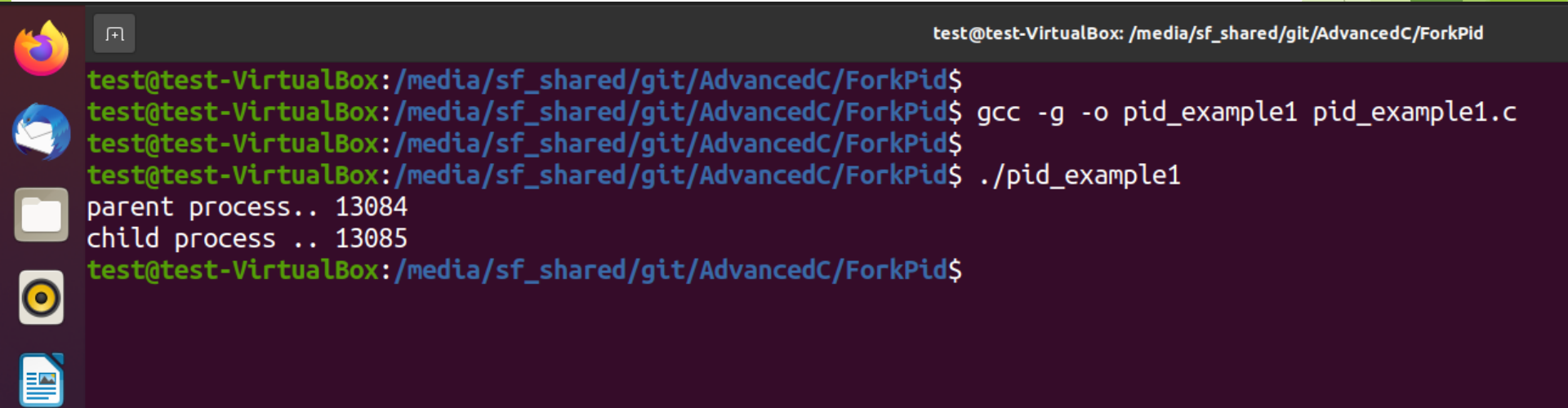
Example 1

```
#include <stdio.h>
#include <unistd.h>
int main (void) {
    printf("I am  process %ld\n", (long)getpid());
    printf("My parent id is %ld\n", (long)getppid());
    return 0;
}
```

Copyright ©: Nahrstedt, Angrave, Abdelzaher


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkPid
#include <stdio.h>
#include <unistd.h>
int main (void) {
printf("I am process %ld\n", (long)getpid());
printf("My parent id is %ld\n", (long)getppid());
return 0;
}
test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkPid
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$ gcc -g -o pid_example pid_example.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$ ./pid_example
I am process 13010
My parent id is 5174
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$


test@test-VirtualBox:
#include <stdio.h>
#include <unistd.h>
int main()
{
pid_t pid;
pid = fork();
if (pid < 0) {
printf("failed to fork..\n");
} else if (pid == 0) {
printf("child process .. %d\n", getpid());
} else if (pid > 0) {
printf("parent process.. %d\n", getpid());
}
sleep(2);
return 0;
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkPid
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$ gcc -g -o pid_example1 pid_example1.c
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$ ./pid_example1
parent process.. 13084
child process .. 13085
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/ForkPid$

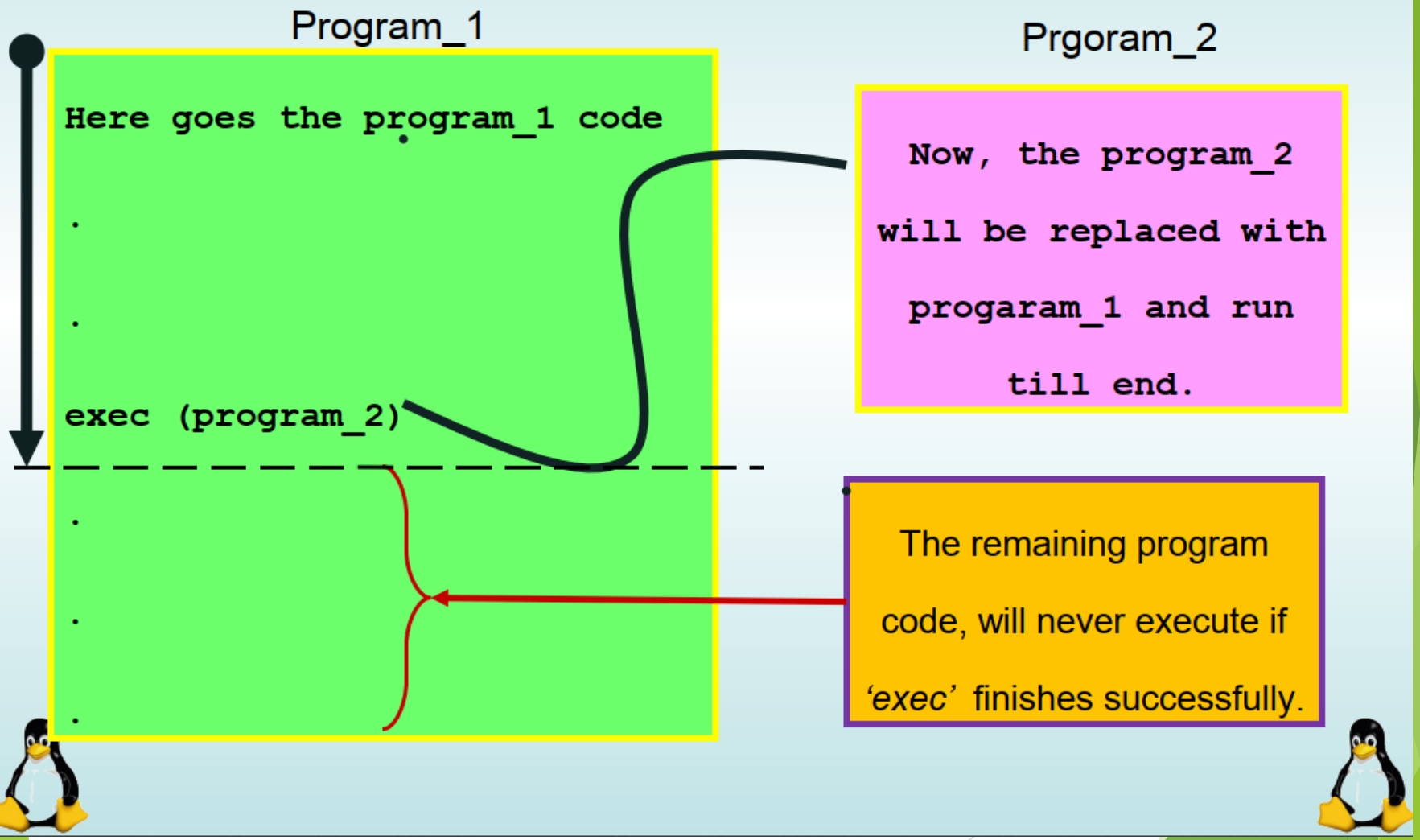
Creating Processes in Linux

- The *exec* family, vary slightly in their capabilities and the way of calling:

  - Functions containing the letter '*p*' in their name, accept a program name and search it in current execution PATH.
  - Those who contain the letter '*v*' or '*l*' in their name, accept the argument list as an array or list for the new program.
  - Those who contain the letter '*e*' in their name, accept an array of environment variables.

exec replaces the calling process with another one, so it will never return a value on success but on failure, returns -1.

Creating Processes in Linux



Creating Processes in Linux

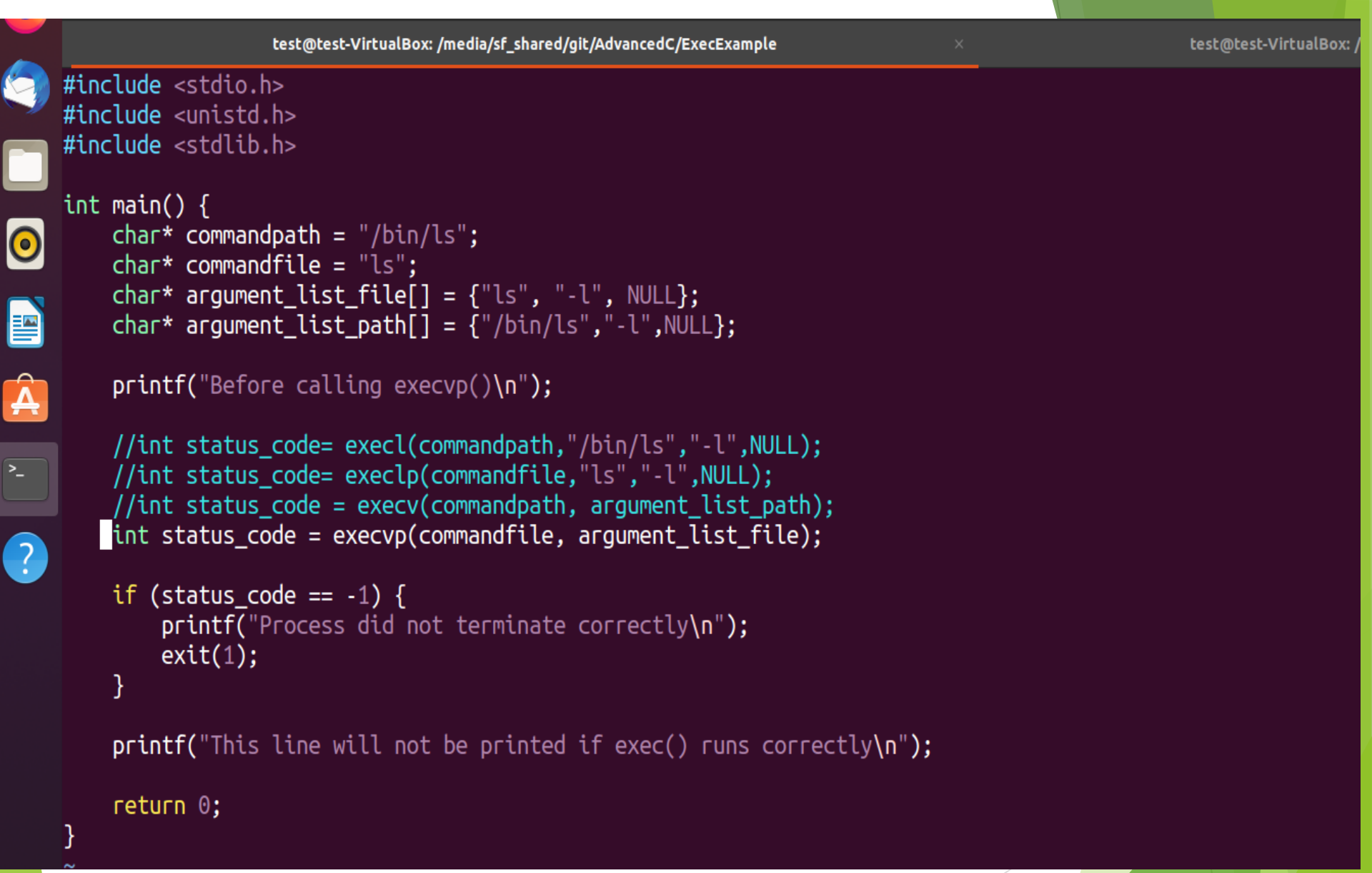
- All of the *exec* family of functions, use just one system call: *execve()*

- execl() functions are variadic functions.

- When calling exec, remember that almost all Linux applications, use argv[0] as their binary image name.

- When using *exec* family, the new process does not have the previous' signal handlers and other stuff.

- The new process has the same values for its PID, PPID, priority and permissions.


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ExecExample
test@test-VirtualBox: /
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
char* commandpath = "/bin/ls";
char* commandfile = "ls";
char* argument_list_file[] = {"ls", "-l", NULL};
char* argument_list_path[] = {"/bin/ls","-l",NULL};
printf("Before calling execvp()\n");
//int status_code= execl(commandpath,"/bin/ls","-l",NULL);
//int status_code= execlp(commandfile,"ls","-l",NULL);
//int status_code = execv(commandpath, argument_list_path);
int status_code = execvp(commandfile, argument_list_file);
if (status_code == -1) {
printf("Process did not terminate correctly\n");
exit(1);
}
printf("This line will not be printed if exec() runs correctly\n");
return 0;
}

Process Termination

Normal exit (voluntary)

- Returning zero from main()
- exit(0)

Error exit (voluntary)

- exit(1)

Fatal error (involuntary)

- Divide by 0, seg fault, exceeded resources

Killed by another process (involuntary)

- Signal: kill(procID)

Process Operations: Termination

When a child process terminates:

- Open files are flushed and closed
- tmp files are deleted
- Child's resources are de-allocated
  - File descriptors, memory, semaphores, file locks, …
- Parent process is notified via signal SIGCHLD
- Exit status is available to parent via wait()

Process Hierarchies

- Parent creates a child process, a child process can create its own child processes
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

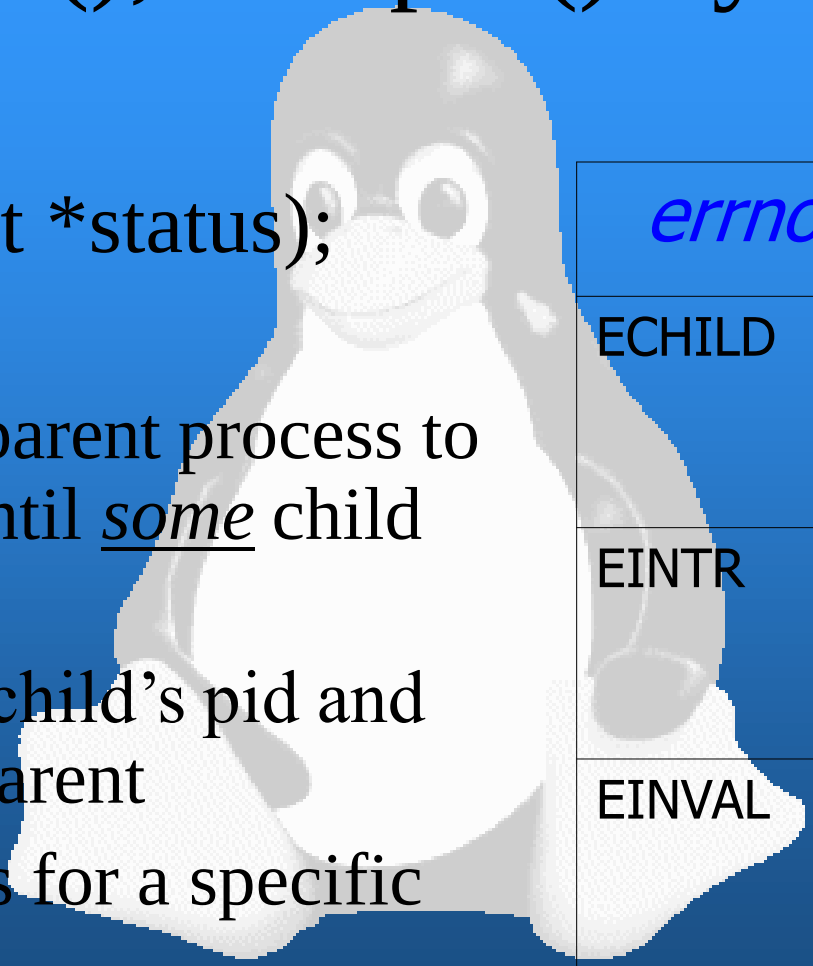
wait(), waitpid() System Calls

pid_t wait(int *status);

wait() causes parent process to wait (block) until *some* child finishes

wait() returns child's pid and exit status to parent

waitpid() waits for a specific child



errno	Cause
ECHILD	Caller has no unwaited-for children
EINTR	Function was interrupted by signal
EINVAL	Options parameter of waitpid was invalid

wait(), waitpid() System Calls

```
pid_t wait(int *status);
```

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie“ state.

If a child has already terminated, then the call returns immediately. Otherwise it blocks until either a child terminates or a signal handler interrupts the call. A child that has terminated and which has not yet been waited upon by this system call (or waitpid) is termed waitable.

wait(), waitpid() System Calls

If status is not NULL, wait() stores status information in the int to which it points. This integer can be inspected with specific macros (see man pages):

WIFEXITED(status)

returns true if the child terminated normally, that is, by calling exit, or by returning from main().

WEXITSTATUS(status)

returns the exit status of the child. This consists of the least significant 8 bits of the status argument that the child specified in a call to exit or as the argument for a return statement in main(). This macro should only be employed if WIFEXITED returned true.

wait() & “zombie”

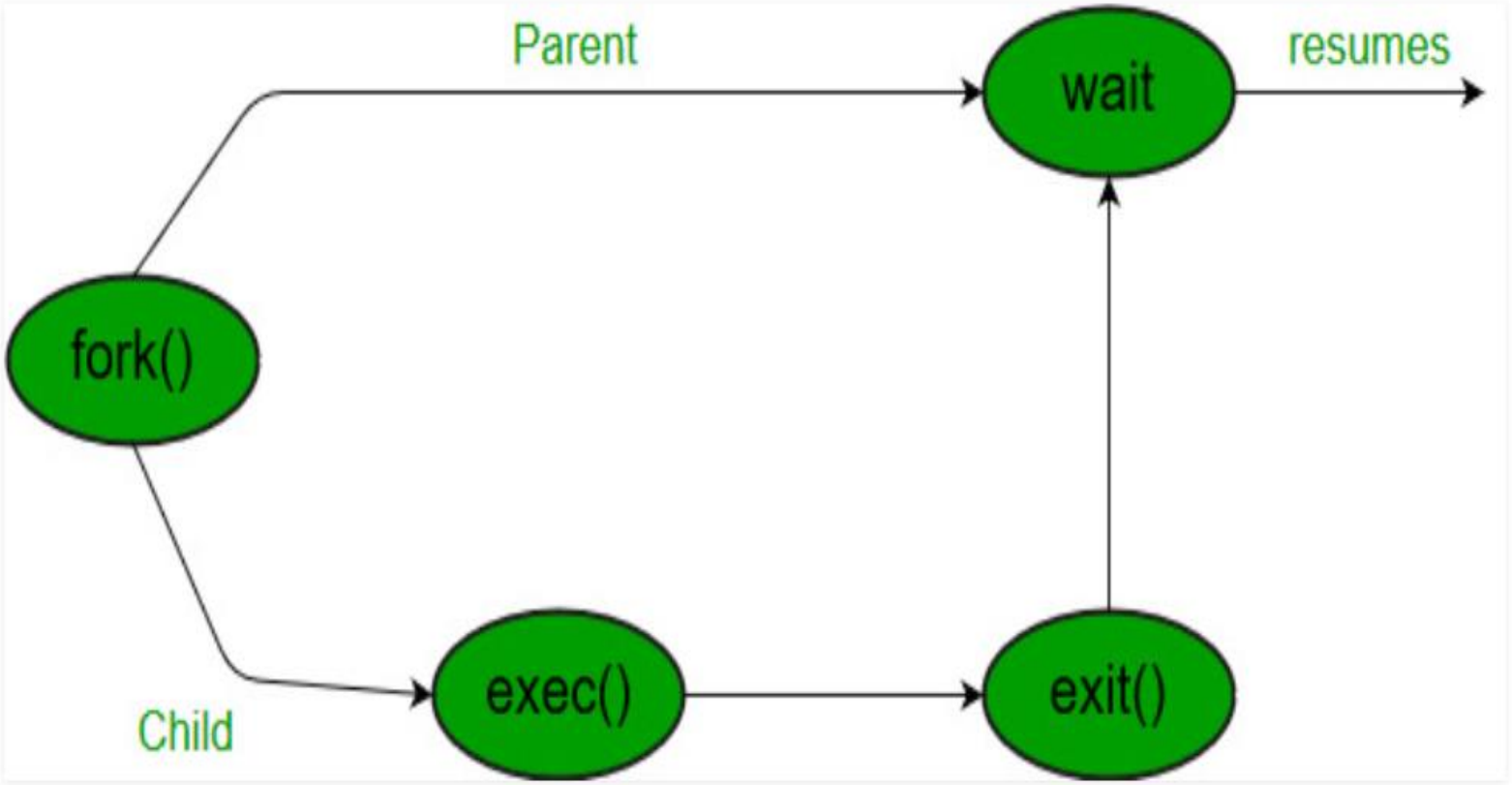
A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child.

As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes.

If a parent process terminates, then its "zombie" children (if any) are adopted by init(8), which automatically performs a wait to remove the zombies.

Waiting for a child to finish (Try “man –s 2 wait”)

```
#include <errno.h>
#include <sys/wait.h>

pid_t childpid;

childpid = wait(NULL);
if (childpid != -1)
    printf(“waited for child with pid %ld\n”, childpid);
```

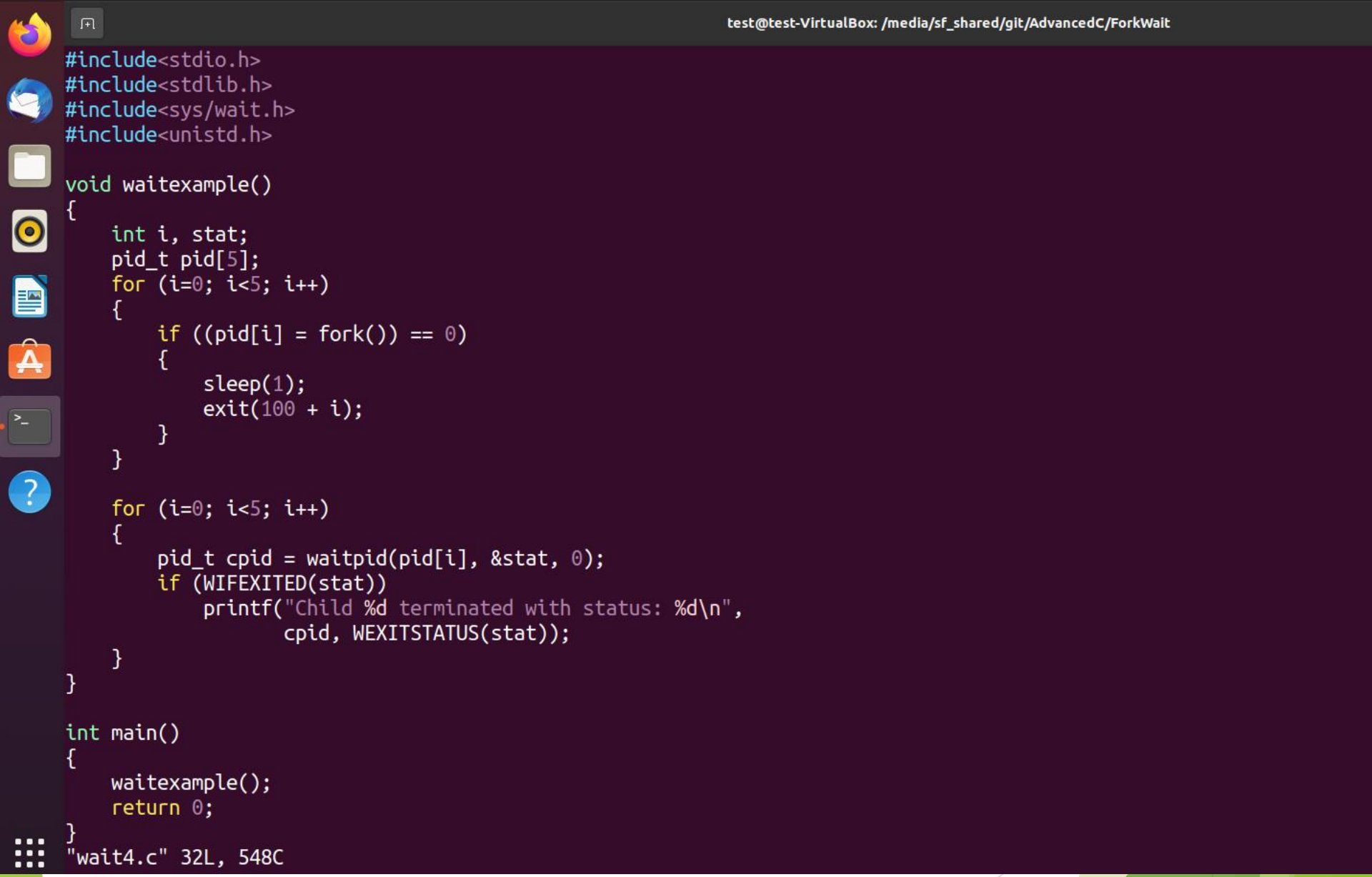

Parent
wait
resumes
fork()
exec()
exit()
Child

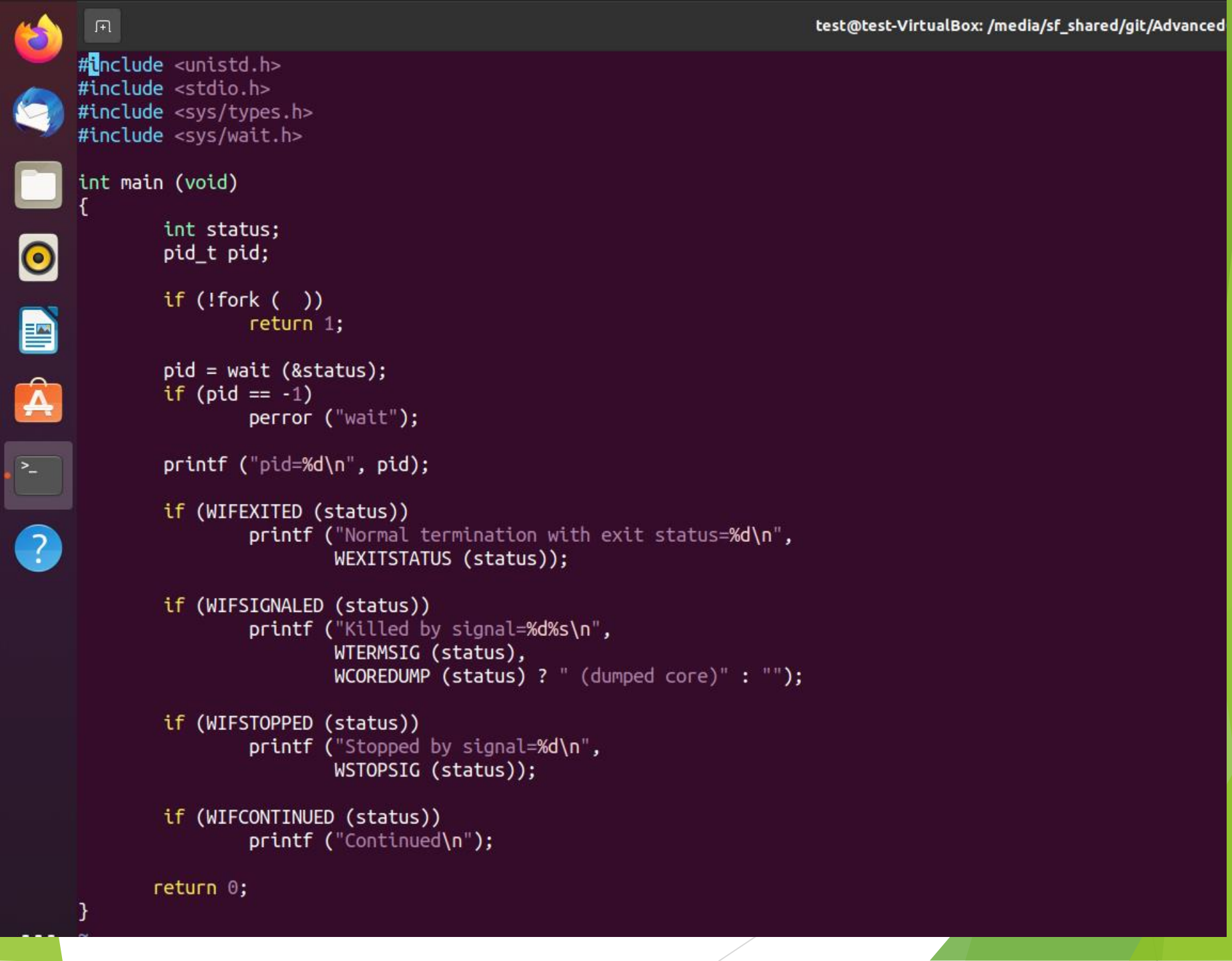

test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkWait
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
pid_t cpid;
if (fork()== 0)
exit(0); /* terminate child */
else
cpid = wait(NULL); /* reaping parent */
printf("Parent pid = %d\n", getpid());
printf("Child pid = %d\n", cpid);
return 0;
}


test@test-VirtualBox: /m
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
if (fork()== 0)
printf("hello from child\n");
else
{
printf("hello from parent\n");
wait(NULL);
printf("child has terminated\n");
}
printf("Bye\n");
return 0;
}
~
~

test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkWait

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

void waitexample()
{
    int stat;

    if (fork() == 0)
        exit(1);
    else
        wait(&stat);
    if (WIFEXITED(stat))
        printf("Exit status: %d\n", WEXITSTATUS(stat));
    else if (WIFSIGNALED(stat))
        psignal(WTERMSIG(stat), "Exit signal");
}

int main()
{
    waitexample();
    return 0;
}
~
```


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/ForkWait
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<unistd.h>
void waitexample()
{
int i, stat;
pid_t pid[5];
for (i=0; i<5; i++)
{
if ((pid[i] = fork()) == 0)
{
sleep(1);
exit(100 + i);
}
}
for (i=0; i<5; i++)
{
pid_t cpid = waitpid(pid[i], &stat, 0);
if (WIFEXITED(stat))
printf("Child %d terminated with status: %d\n",
cpid, WEXITSTATUS(stat));
}
}
int main()
{
waitexample();
return 0;
}
"wait4.c" 32L, 548C


test@test-VirtualBox: /media/sf_shared/git/Advanced
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
int main (void)
{
int status;
pid_t pid;
if (!fork ( ))
return 1;
pid = wait (&status);
if (pid == -1)
perror ("wait");
printf ("pid=%d\n", pid);
if (WIFEXITED (status))
printf ("Normal termination with exit status=%d\n",
WEXITSTATUS (status));
if (WIFSIGNALED (status))
printf ("Killed by signal=%d%s\n",
WTERMSIG (status),
WCOREDUMP (status) ? " (dumped core)" : "");
if (WIFSTOPPED (status))
printf ("Stopped by signal=%d\n",
WSTOPSIG (status));
if (WIFCONTINUED (status))
printf ("Continued\n");
return 0;
}

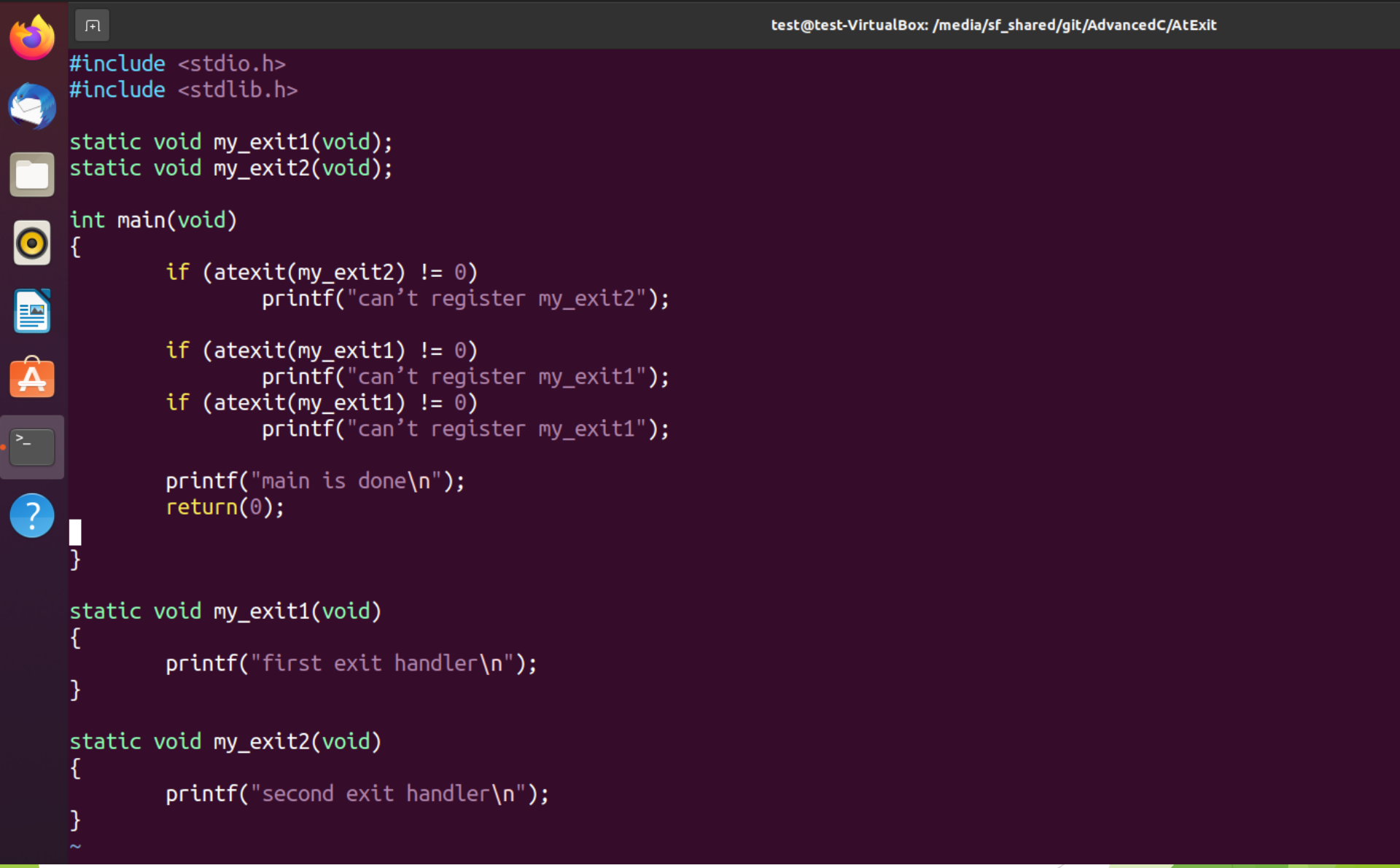
POSIX 1003.1-2001 defines, and Linux implements, the `atexit( )` library call, used to register functions to be invoked on process termination:

```
#include <stdlib.h>

int atexit (void (*function)(void));
```

A successful invocation of `atexit( )` registers the given function to run during normal process termination; i.e., when a process is terminated via either `exit( )` or a return from `main( )`. If a process invokes an exec function, the list of registered functions is cleared (as the functions no longer exist in the new process' address space). If a process terminates via a signal, the registered functions are not called.

The given function takes no parameters, and returns no value. A prototype has the form:

```
void my_function (void);
```


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/AtExit
#include <stdio.h>
#include <stdlib.h>
static void my_exit1(void);
static void my_exit2(void);
int main(void)
{
if (atexit(my_exit2) != 0)
printf("can't register my_exit2");
if (atexit(my_exit1) != 0)
printf("can't register my_exit1");
if (atexit(my_exit1) != 0)
printf("can't register my_exit1");
printf("main is done\n");
return(0);
}
static void my_exit1(void)
{
printf("first exit handler\n");
}
static void my_exit2(void)
{
printf("second exit handler\n");
}

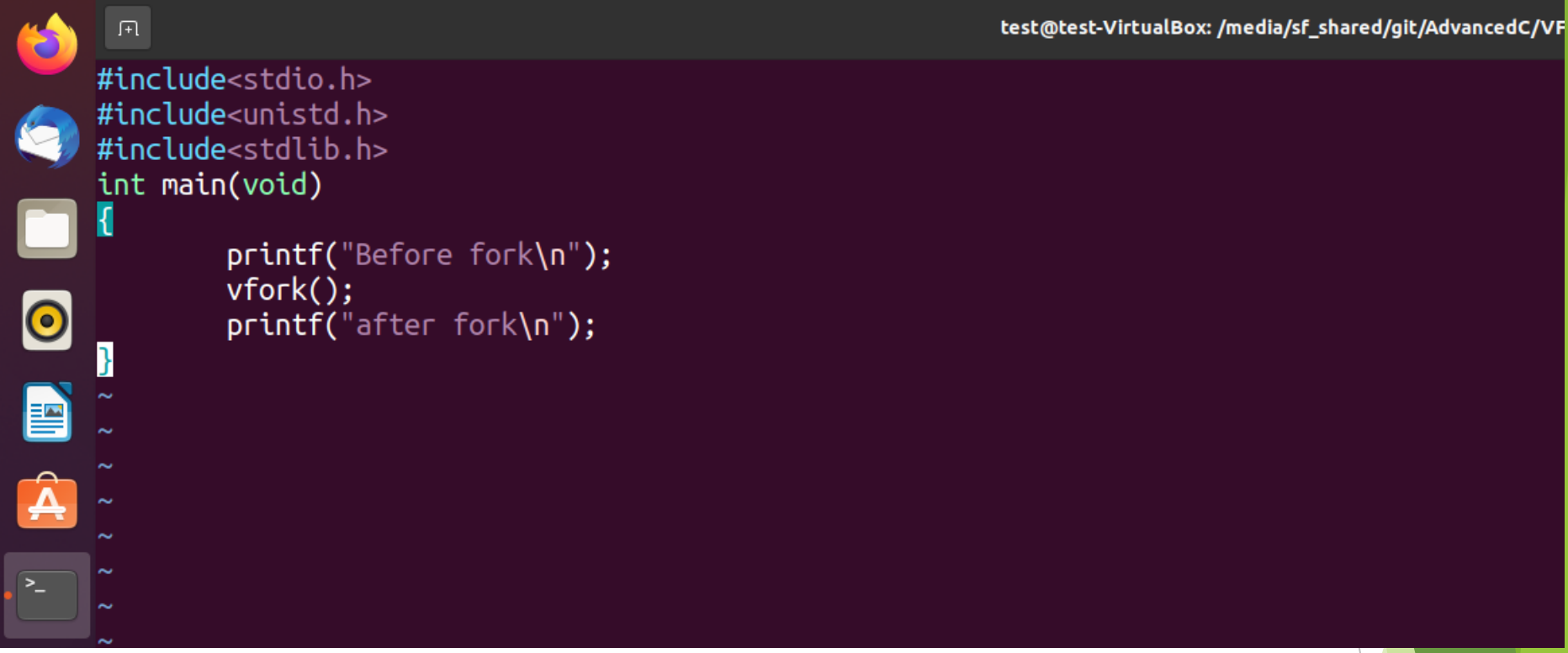

test@test-VirtualBox: /media
#include <stdio.h>
#include <stdlib.h>
void out (void)
{
printf ("atexit( ) succeeded!\n");
}
int main (void)
{
if (atexit (out))
fprintf(stderr, "atexit( ) failed!\n");
return 0;
}

vfork

- Creates a new processes with the express purpose of exec-ing a new program
- New child process runs in parent's address space until exec or exit is called
- vfork guarantees that the child will run before the parent until exec or exit call is reached

Differences between fork() and vfork()

	fork()	vfork()
Address space	Both the child and parent process will have different address space	Both child and parent process share the same address space
Modification in address space	Any modification done by the child in its address space is not visible to parent process as both will have separate copies	Any modification by child process is visible to both parent and child as both will have same copies
CoW(copy on write)	This uses copy-on-write.	Vfork doesn't use CoW
Execution summary	Both parent and child executes simultaneously	Parent process will be suspended until child execution is completed.
Outcome of usage	Behaviour is predictable	Behaviour is not predictable


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/VF
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(void)
{
printf("Before fork\n");
vfork();
printf("after fork\n");
}


test@test-VirtualBox: /media/sf_shared/git/AdvancedC/VFork
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n =10;
pid_t pid = vfork(); //creating the child process
if (pid == 0) //if this is a chile process
{
n=50;
printf("Child process started\n");
exit(0);
}
else//parent process execution
{
printf("Now i am coming back to parent process\n");
}
printf("value of n: %d \n",n); //sample printing to check "n" value
return 0;
}

- **Clone process** is created using primitive “clone,” by duplicating its parent process
  - Allows both processes to share same segment of code and data
  - Modification of one is visible to other, which is unlike classical processes
- Ability to clone processes brings possibility of implementing servers in which several threads may be executing

A *daemon* is a process that runs in the background, not connecting to any controlling terminal. Daemons are normally started at boot time, are run as root or some other special user (such as *apache* or *postfix*), and handle system-level tasks. As a convention, the name of a daemon often ends in *d* (as in *crond* and *sshd*), but this is not required, or even universal.

The name derives from Maxwell's demon, an 1867 thought experiment by the physicist James Maxwell. Daemons are also supernatural beings in Greek mythology, existing somewhere between humans and the gods and gifted with powers and divine knowledge. Unlike the demons of Judeo-Christian lore, the Greek daemon need not be evil. Indeed, the daemons of mythology tended to be aides to the gods, performing tasks that the denizens of Mount Olympus found themselves unwilling to do—much as Unix daemons perform tasks that foreground users would rather avoid.

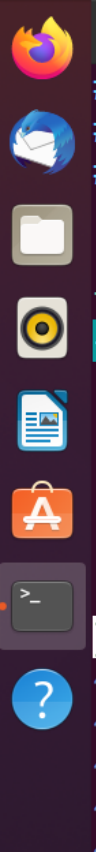
A daemon has two general requirements: it must run as a child of init, and it must not be connected to a terminal.

In general, a program performs the following steps to become a daemon:

1. Call `fork( )`. This creates a new process, which will become the daemon.
2. In the parent, call `exit( )`. This ensures that the original parent (the daemon's grandparent) is satisfied that its child terminated, that the daemon's parent is no longer running, and that the daemon is not a process group leader. This last point is a requirement for the successful completion of the next step.
3. Call `setsid( )`, giving the daemon a new process group and session, both of which have it as leader. This also ensures that the process has no associated controlling terminal (as the process just created a new session, and will not assign one).
4. Change the working directory to the root directory via `chdir( )`. This is done because the inherited working directory can be anywhere on the filesystem. Daemons tend to run for the duration of the system's uptime, and you don't want to keep some random directory open, and thus prevent an administrator from unmounting the filesystem containing that directory.
5. Close all file descriptors. You do not want to inherit open file descriptors, and, unaware, hold them open.
6. Open file descriptors 0, 1, and 2 (standard in, standard out, and standard error) and redirect them to */dev/null*.

test@test-VirtualBox: /media/sf_shared/git/AdvancedC/DaemonExa

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/fs.h>

int main (void)
{
        pid_t pid;
        int i;

        /* create new process */
        pid = fork (  );
        if (pid == -1)
                return -1;
        else if (pid != 0)
                exit (EXIT_SUCCESS);

        /* create new session and process group */
        if (setsid (  ) == -1)
                return -1;

        /* set the working directory to the root directory */
        if (chdir ("/") == -1)
                return -1;

        /* close all open files--NR_OPEN is overkill, but works */
        for (i = 0; i < 3; i++)
                close (i);

        /* redirect fd's 0,1,2 to /dev/null */
        open ("/dev/null", O_RDWR);     /* stdin */
        dup (0);                        /* stdout */
        dup (0);                        /* stderror */

        /* do its daemon thing... */

        return 0;
}
```



test@test-VirtualBox: /media/sf_shared/git/AdvancedC/DaemonExample

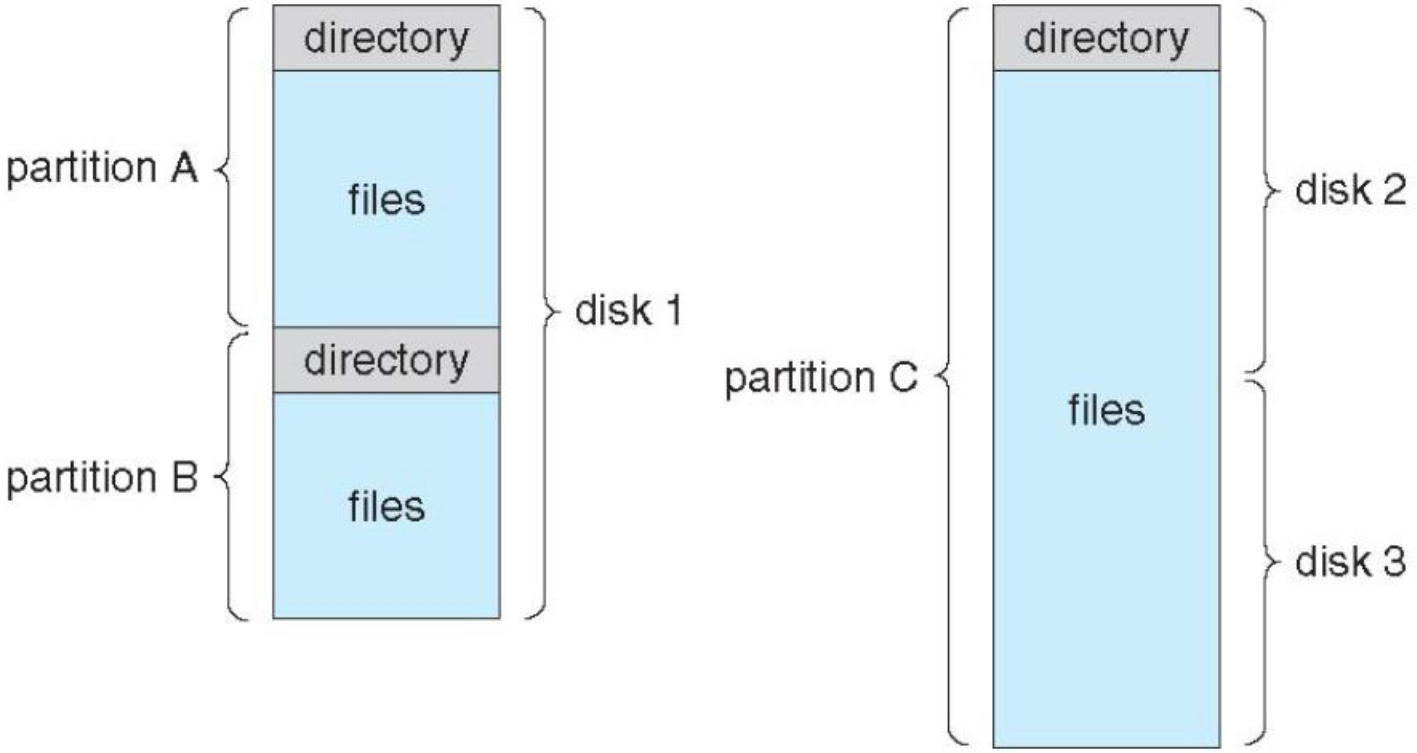
```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>


int main(int argc, char *argv[])
{
        int nochdir=1;
        int noclose=0;

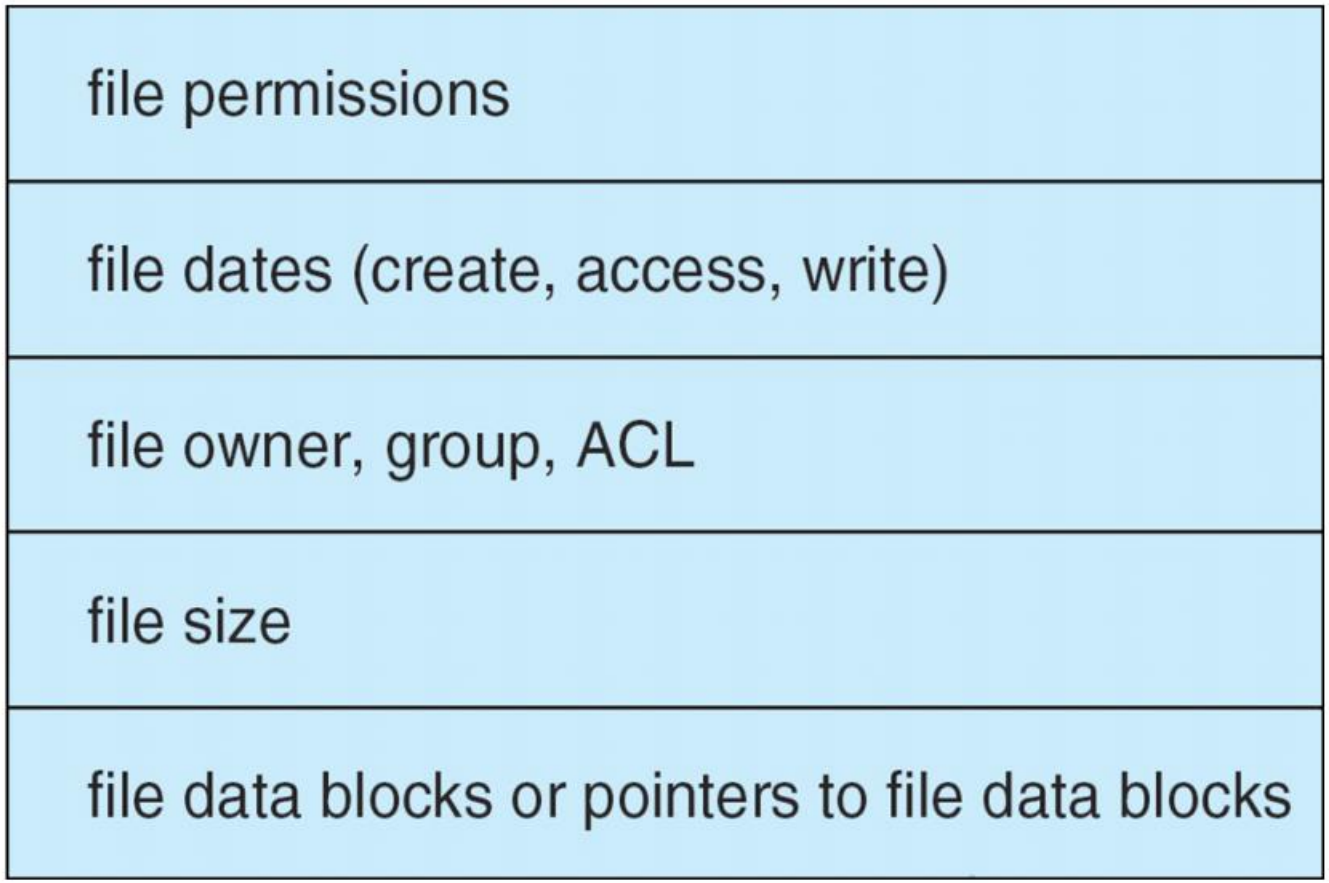
        daemon(nochdir,noclose);

        return 0;
}
~
~
~
~
```

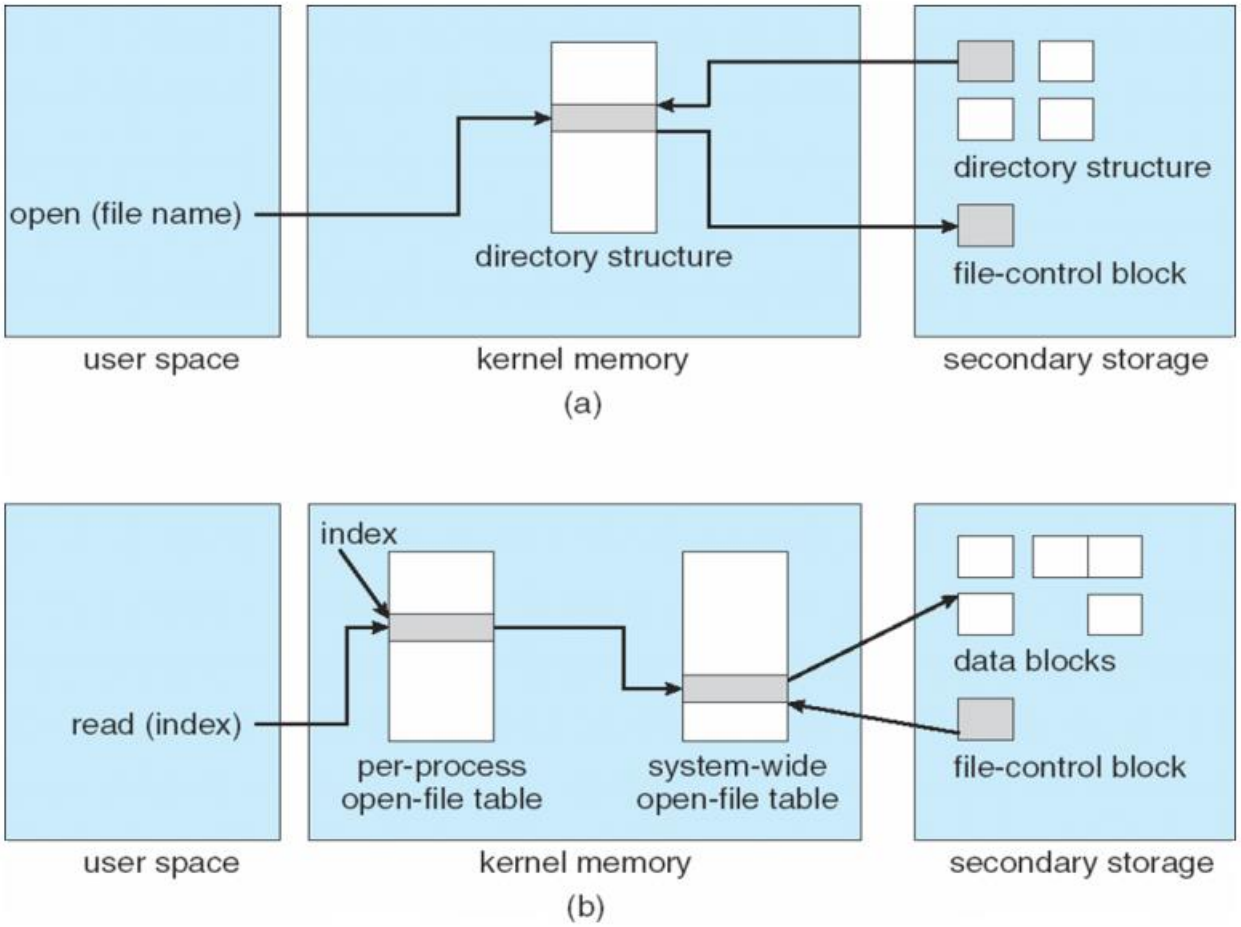
A Typical File-system Organization



A Typical File Control Block



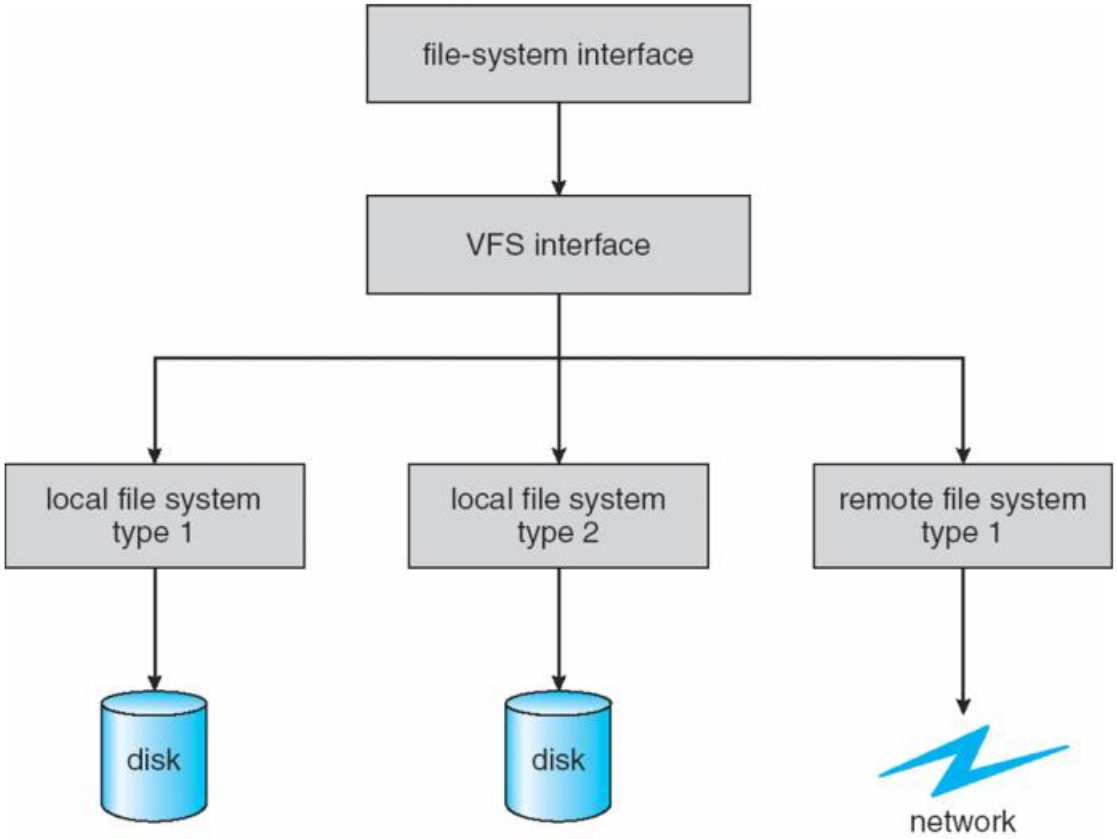
In-Memory File System Structures



Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.

Schematic View of Virtual File System



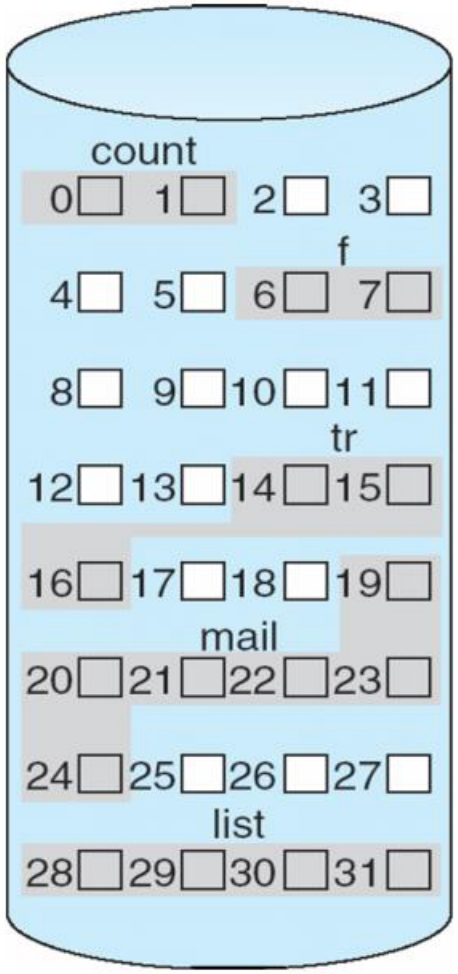
Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
- Contiguous allocation
- Linked allocation
- Indexed allocation

Contiguous Allocation

- Each file occupies a set of contiguous blocks on the disk
- Simple – only starting location (block #) and length (number of blocks) are required
- Random access
- Wasteful of space (dynamic storage-allocation problem)
- Files cannot grow

Contiguous Allocation of Disk Space

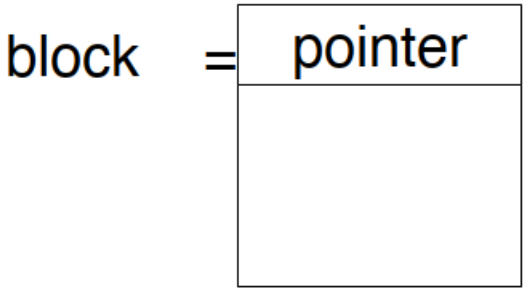


directory

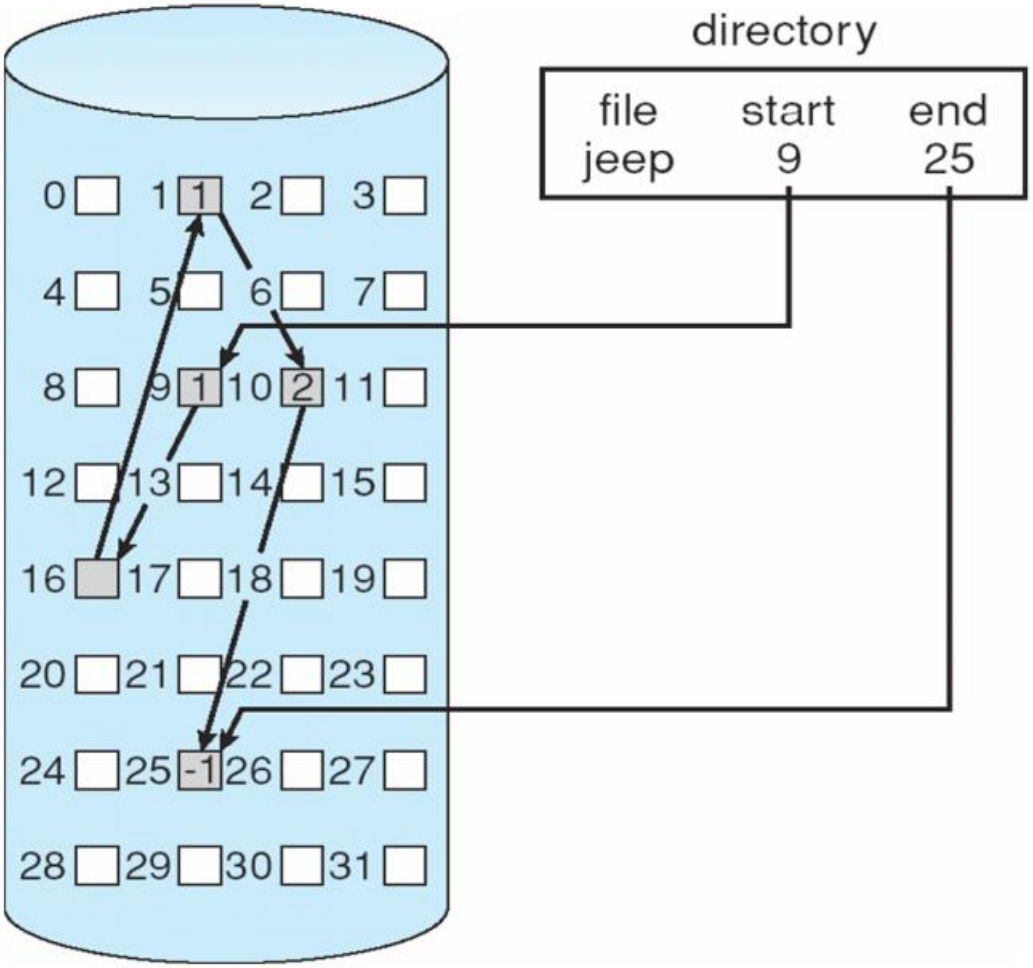
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked Allocation

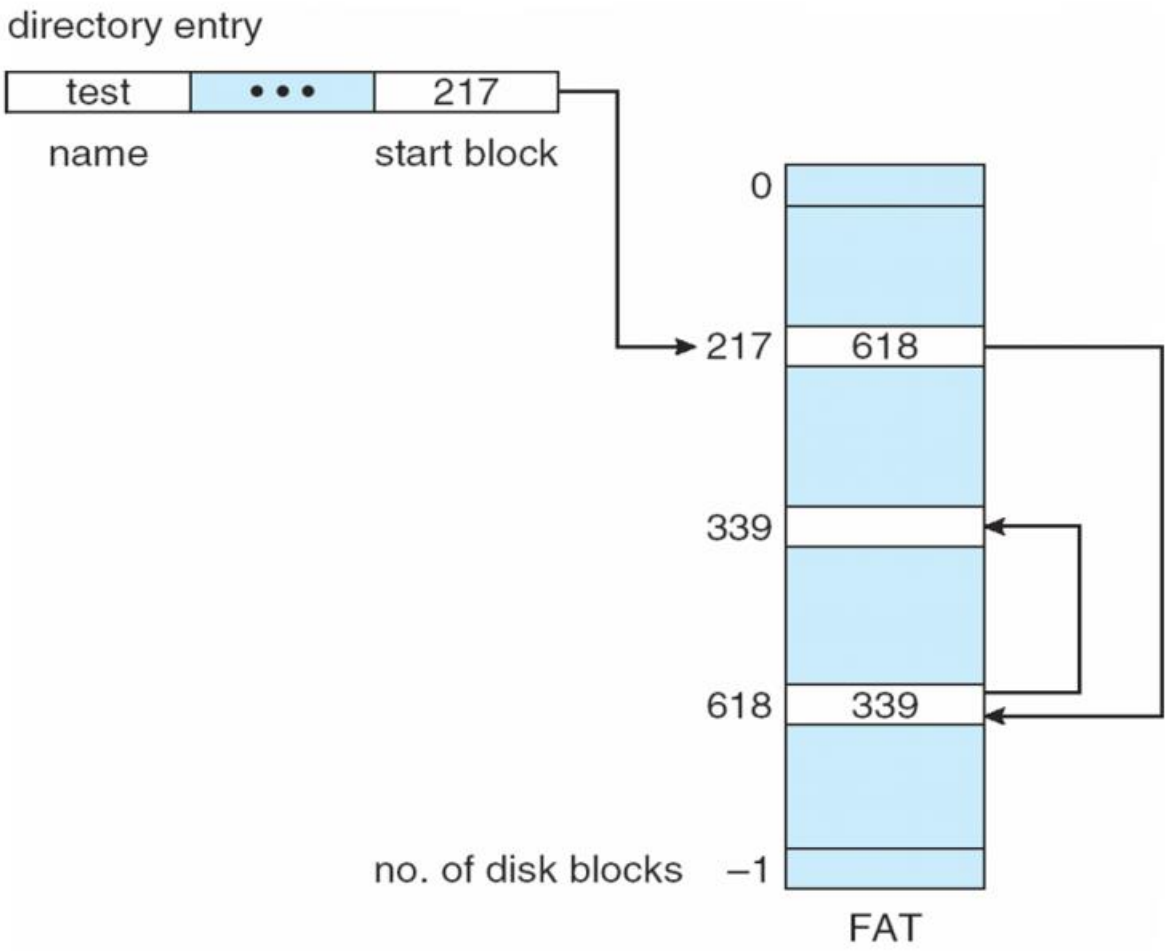
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk.



Linked Allocation

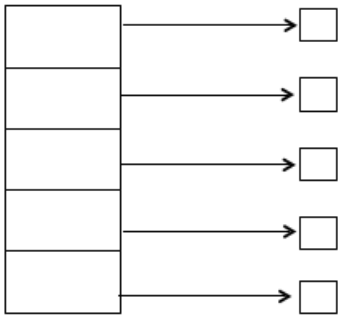


File-Allocation Table



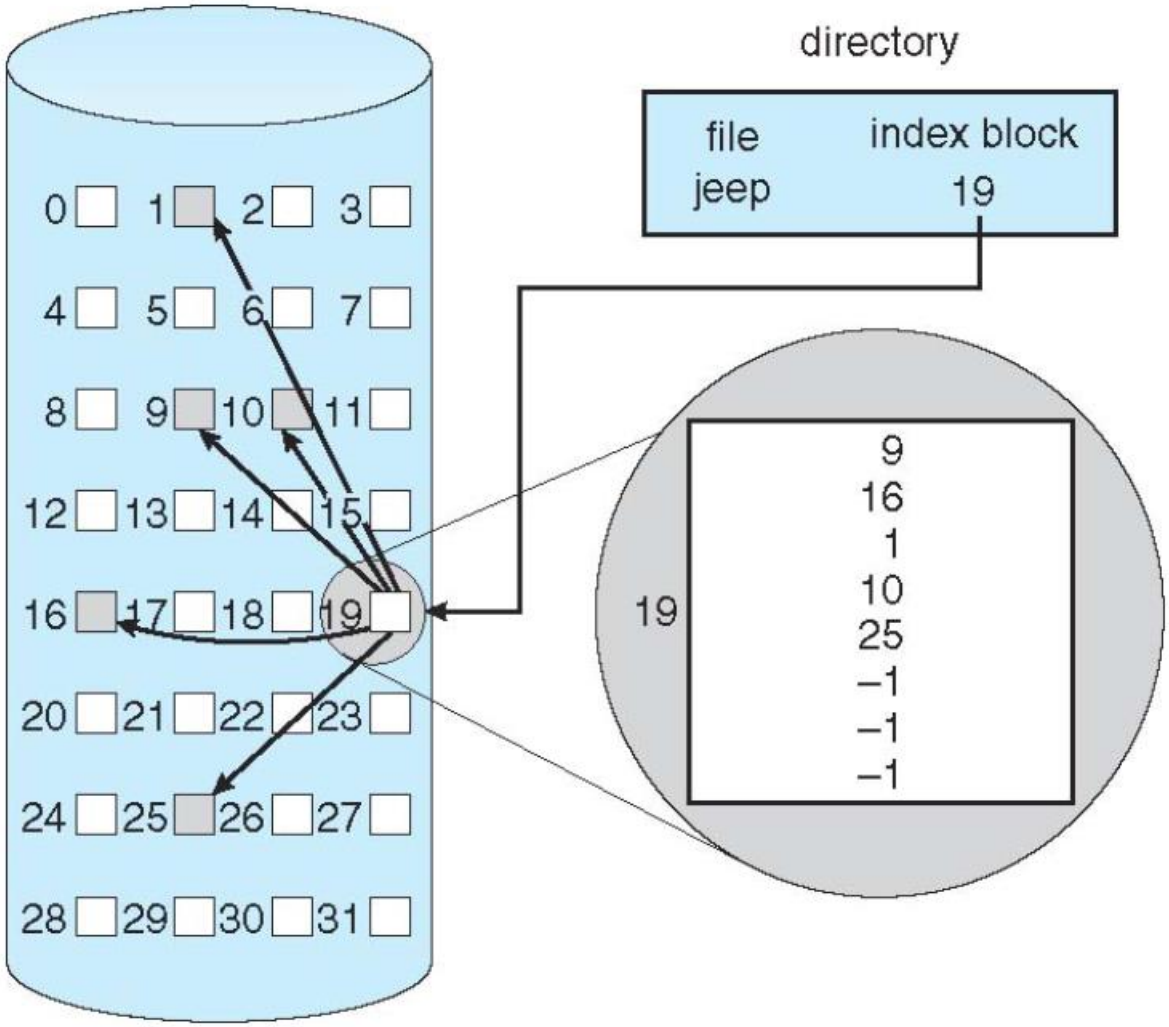
Indexed Allocation

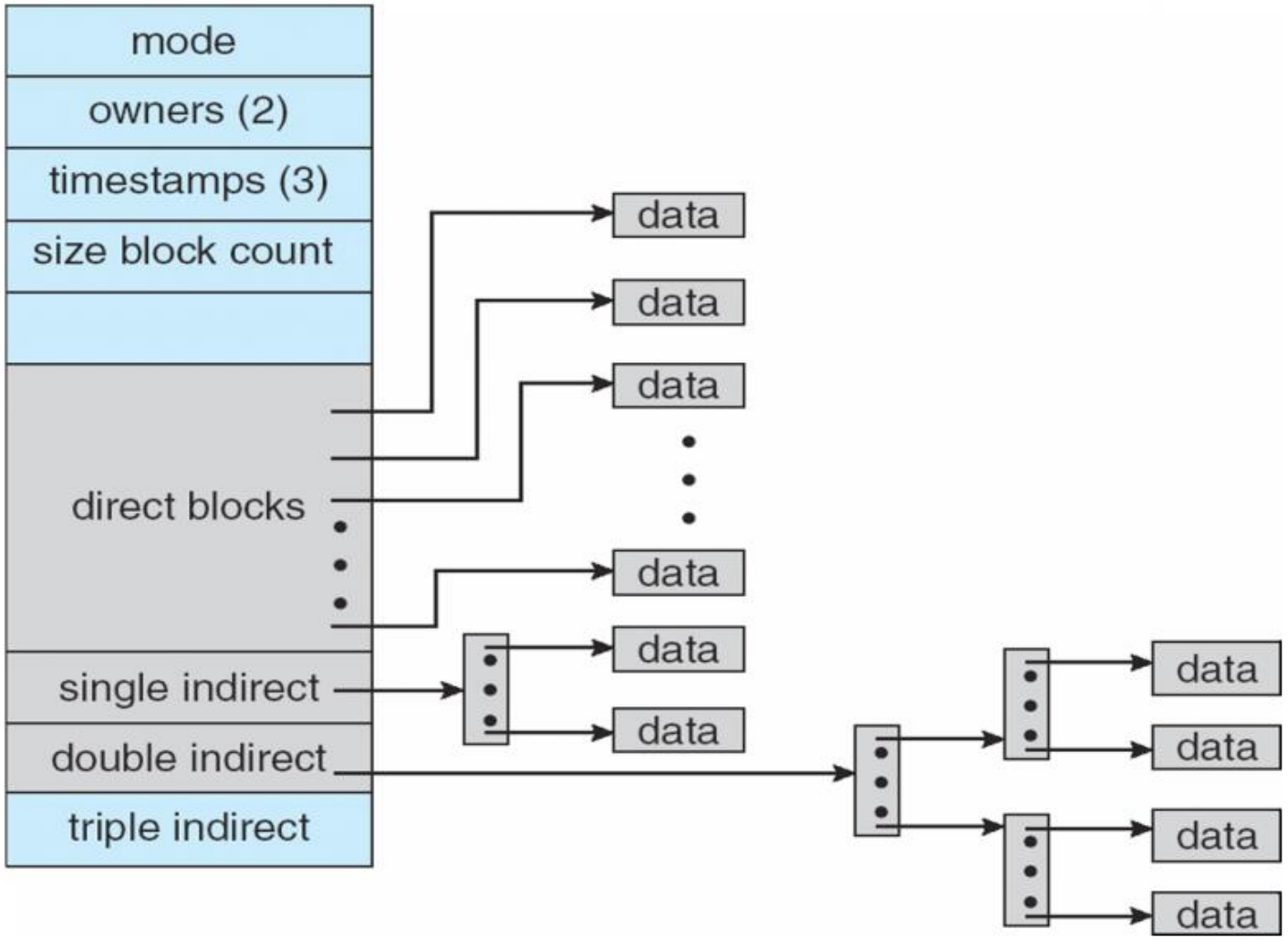
- Brings all pointers together into the **index block**
- Logical view



index table

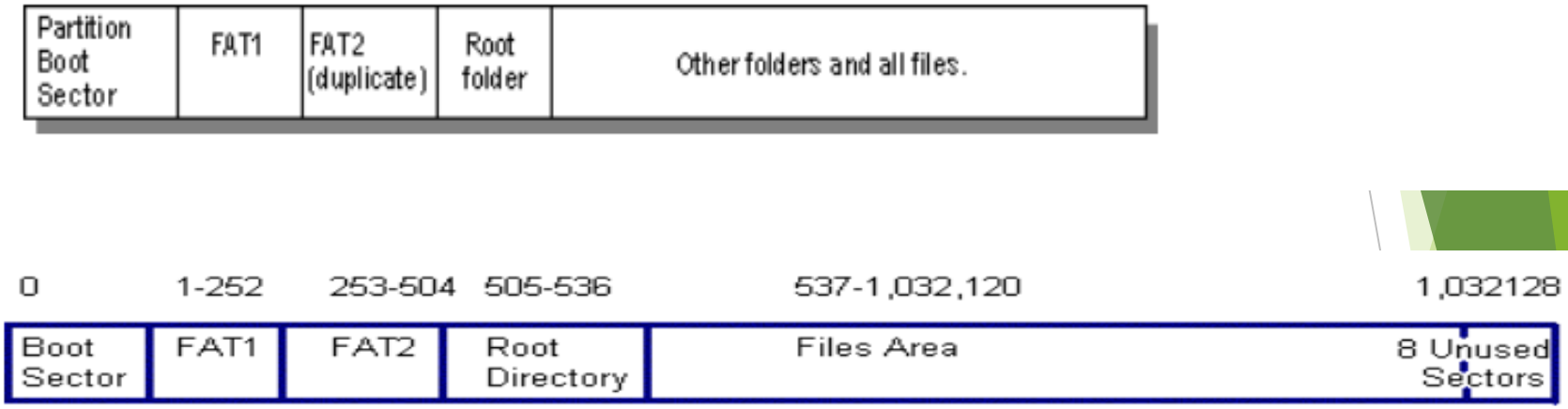
Example of Indexed Allocation




mode
owners (2)
timestamps (3)
size block count
direct blocks
single indirect
double indirect
triple indirect
data
data
data
data
data
data
data
data
data
data

Structure of a FAT Volume

The next figure illustrates how the FAT file system organizes a volume.



FAT Boot Sector and BPB

Offset	Length of Field	Typical Value	Meaning
0x00	3B	eb 34 90	Jump Instruction
0x03	8B	IBM 3.3	OEM Manufacturer
0x0b	25B		BIOS Parameter Block
0x24	26B		Extended Bios Parameter Block
0x3e	448 B		Bootstrap Code
0x1fe	2B	55 aa	End of Sector Marker

Offset	Length of Field	Typical Value(in hex)	Meaning
0x0b, 11	2B	00 02	The number of bytes, in big-endian. The typical value of 0002 translates to 0x0200 = 512
0x0d, 13	1B	01	Number of Sectors per Cluster.
0x0e, 14	2B	0100	Number of Reserved Sectors. The number is at least 1. If the number is larger, then the bootstrap code does not fit in the allotted space in the partition boot sector.
0x10, 16	1B	02	Number of File Allocation Tables, typically 2. (This provides redundancy against corruption.)
0x11, 17	2B	e0 00	Root Entries. The total number file name entries that can be stored in the root folder of the volume. For FAT12 and FAT16 volumes, this value should always specify a count that when multiplied by 32 results in an even multiple of BPB_BytsPerSec. FAT16 should use 512. For FAT32 volumes, this number is set to zero.
0x13, 19	2B	40 0b	The total number of small sectors. Here, this number is 2880, the value for a 1.4MB floppy.
0x15, 21	1B	f0	Media Type: f0 removable, f8 fixed media, i.e. hard disk.
0x16, 22	2B	09 00	Sectors per file allocation table. This is useful to determine the location of the root folder. For FAT32, this field should be zero.
0x18, 24	2B	12 00	Sectors per track, here 18. Hence, there must be 80 tracks.
0x1A, 26	2B	02 00	Number of heads, here 2.
0x1C, 28	4B	00 00 00 00	Count of hidden sectors preceding the partition that contains the FAT volume.
0x20, 32	4B	00 00 00 00	Number of sectors. Either this field or the one at 0x13 is zero, depending on whether the number fits in the first field. Thus, for FAT32 the total count of sectors.
0x24, 36	1B	00	Physical disk number. 0 is the A drive disk, hard disks start at 0x80.
0x25, 37	1B	00	Current Head. Unused in FAT file system.
0x26, 38	1B	00	Boot Signature.
0x27, 39	4B		Volume serial number.
0x2b, 43	11B	00 00 00 ... 00	Volume Label
0x36	8B		System ID. either FAT12 or FAT16.

FAT Data Structure

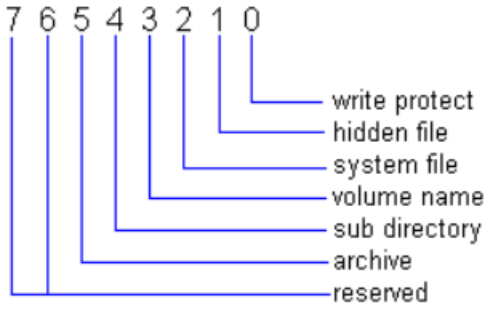
FAT 12	FAT 16	Meaning
000	0000	available for allocation
001	0001	not used
FF0	FFF0-FFF6	reserved
FF7	FFF7	bad cluster, dna
FF8-FFF	FFF8-FFFF	last cluster in file
everything else	everything else	next cluster

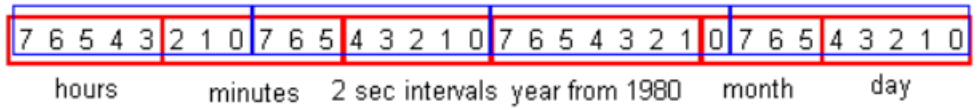
FAT12 & FAT16 Root Directories

Offset	Length of Field	Typical Value	Meaning
0x00	8B	49 4F 20 20 20 20 20 20	File name, padded with spaces
0x08	3B	53 59 53	3B file extension
0x0b	1B	04	File Attribute
0x0c	10B	27 00 00 00 00 00 00 75 2F 00 00	Reserved
0x16	2B	65 59	Time of last change
0x18	2B	18 21	Date of Last Change
0x1a	2B	02 00	First Cluster
0x1c	4B	34 47 03 00	File Size

The first character of the file name is important.

- 0x00 Entry never used
- 0xe5 File is deleted
- 0x2e (A ".") Directory
- 0x05 In a FAT32 entry, 0x05 as the lead character is translated to 0xe5, a Kanji character, so that Japanese language versions work.





Anandkumar

FAT Handson

- dd if=/dev/zero of=test.img count=1 bs=1M
- hexdump -Cv test.img > original.txt
- mkfs.fat test.img
- mount test.img test

```
#include <unistd.h>

ssize_t pread (int fd, void *buf, size_t count, off_t pos);
```

This call reads up to `count` bytes into `buf` from the file descriptor `fd` at file position `pos`.

```
#include <unistd.h>

ssize_t pwrite (int fd, const void *buf, size_t count, off_t pos);
```

This call writes up to `count` bytes from `buf` to the file descriptor `fd` at file position `pos`.

These calls are almost identical in behavior to their non-`p` brethren, except that they completely ignore the current file position; instead of using the current position, they use the value provided by `pos`. Also, when done, they do not update the file position. In other words, any intermixed `read( )` and `write( )` calls could potentially corrupt the work done by the positional calls.

Both positional calls can be used only on seekable file descriptors. They provide semantics similar to preceding a `read( )` or `write( )` call with a call to `lseek( )`, with three differences. First, these calls are easier to use, especially when doing a tricky operation such as moving through a file backward or randomly. Second, they do not update the file pointer upon completion. Finally, and most importantly, they avoid any potential races that might occur when using `lseek( )`. As threads share file descriptors, it would be possible for a different thread in the same program to update the file position after the first thread's call to `lseek( )`, but before its read or write operation executed. Such race conditions can be avoided by using the `pread( )` and `pwrite( )` system calls.

```
ssize_t readv (int fd,
               const struct iovec *iov,
               int count);
```

The `writev( )` function writes at most `count` segments from the buffers described by `iov` into the file descriptor `fd`:

```
#include <sys/uio.h>

ssize_t writev (int fd,
                const struct iovec *iov,
                int count);
```

The `readv( )` and `writev( )` functions behave the same as `read( )` and `write( )`, respectively, except that multiple buffers are read from or written to.

Each `iovec` structure describes an independent disjoint buffer, which is called a *segment*:

- The main advantages offered by readv and writev are:
  - It allows working with non contiguous blocks of data i.e. buffers need ot be part of an array, but separately allocated.
  - The I/O is 'atomic' i.e if we do writev, all the elements in the vector will be written in one contiguous operation, and writes done by other processes will not occur in between them.

Timing results comparing `writev` and other techniques

	Linux (Intel x86)			Mac OS X (PowerPC)		
Operation	User	System	Clock	User	System	Clock
two `write`s	1.29	3.15	7.39	1.60	17.40	19.84
buffer copy, then one `write`	1.03	1.98	6.47	1.10	11.09	12.54
one `writev`	0.70	2.72	6.41	0.86	13.58	14.72

IPC

- Inter Process Communication (IPC) is transfer of data between processes.

- In Linux there are some methods of IPC:

  - Shared Memory.
  - Mapped Memory.
  - Pipe (Named and Unnamed).
  - Socket (Remote, Local).

Pipes

- A pipe is a communicational device that permits unidirectional communication.

- The first data written into pipe is the first one that is read.

- If the writer, writes faster than the reader and pipe is full, the writer blocks.

- If the reader reads tries to read an empty pipe, it blocks.

- You can create pipes using *pipe()*.

Pipes

On success, 0 is returned and on error, -1 is returned and errno is set.

An array of size 2, which contains read and write file descriptors.

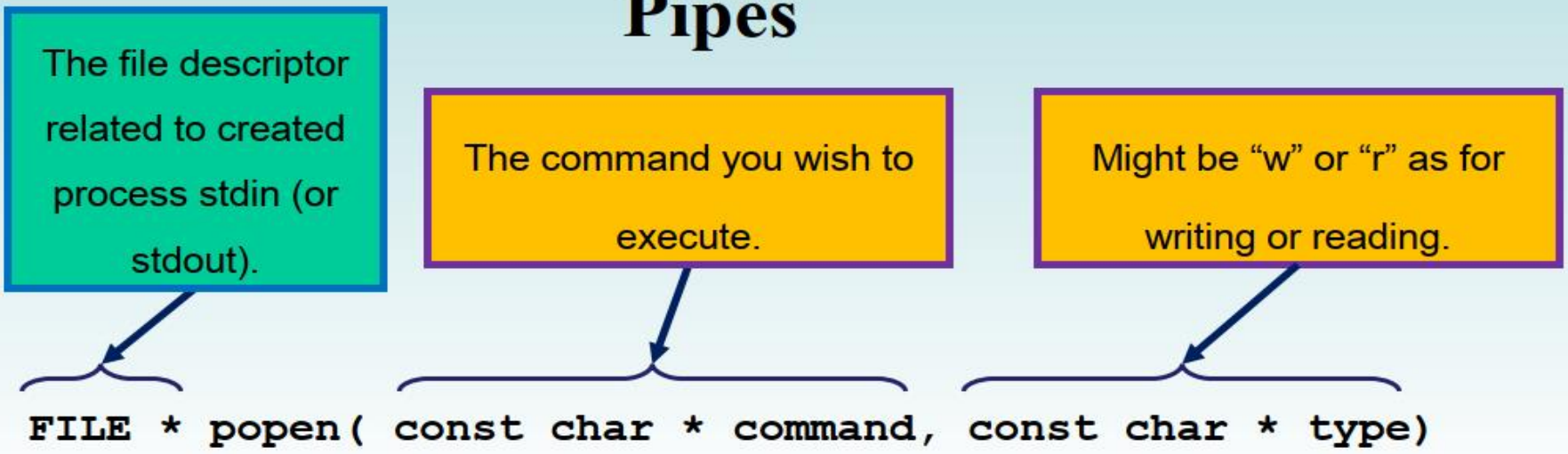
```
int  pipe ( int filedes[2] );
```

- *pipe()* stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

- Read and write file descriptors are available only in calling process and its children.

- You can use pipes to communicate between threads in a process.

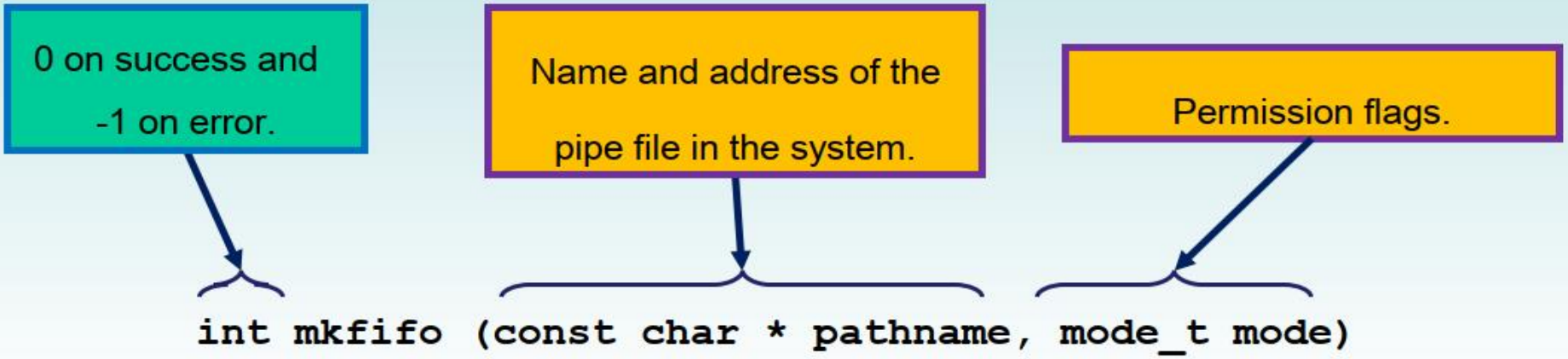
Pipes



- You can use *popen()* to send data to or receive data from a program running in a subprocess.

- After closing the stream (using *pclos()*), *pclose()* waits for the child process to terminate.

Named Pipes



- You can access a named pipe like an ordinary file.

- One program must open it for writing and another for reading.

- In computing, a named pipe (also known as a **FIFO**) is one of the methods for intern-process communication.
  - It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process.
  - A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used.
  - Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.

- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Message Queue

- Another IPC mechanism in Linux.

- Implemented both SYS V type and POSIX type and it's structure is somehow like pipes (FIFO).

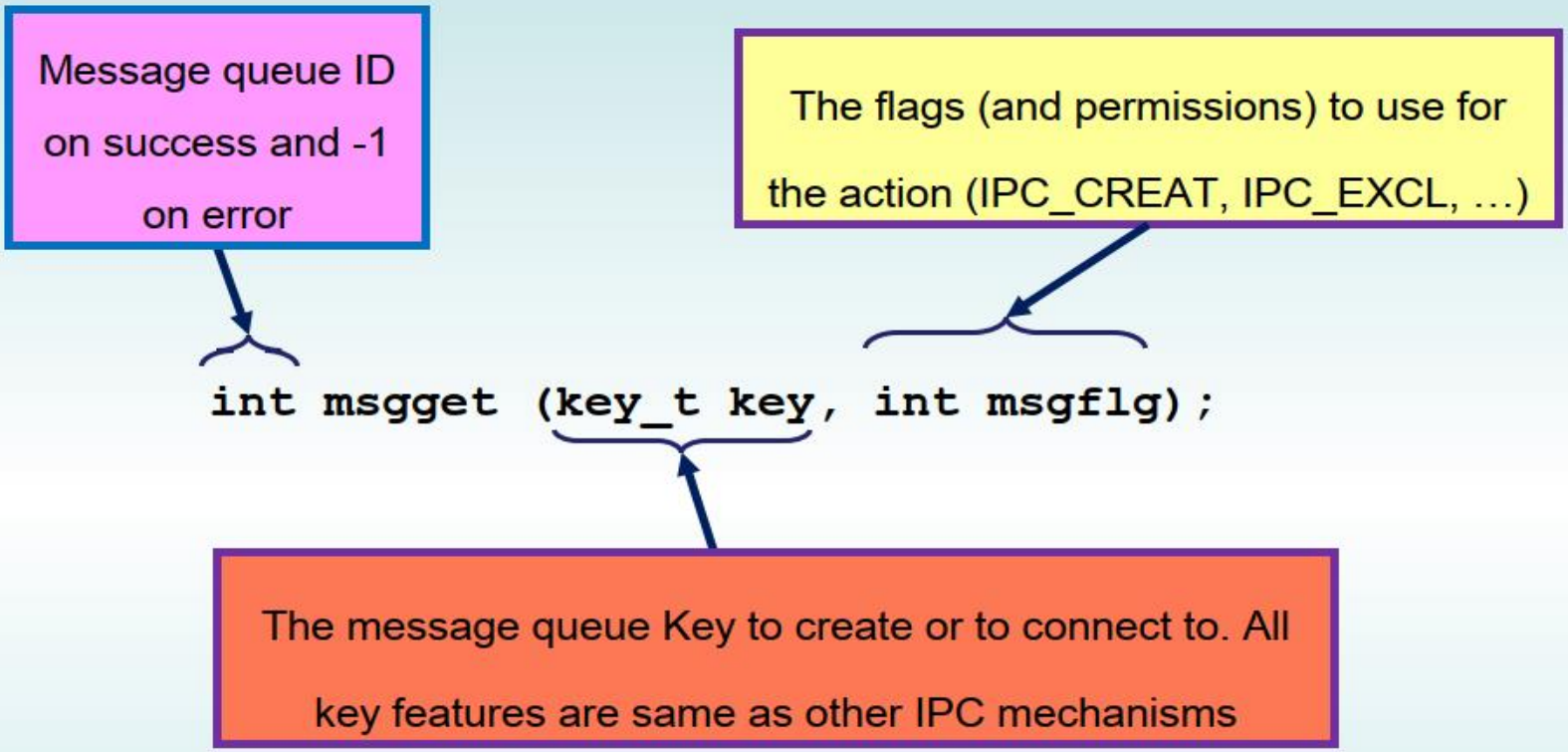
- A process initializes a message queue and itself or other processes can put messages in this queue, knowing the MSQID of this message queue.

- After finishing the job, one process should deallocate the queue.

- If you don't remove a message queue it will remain even after process termination.

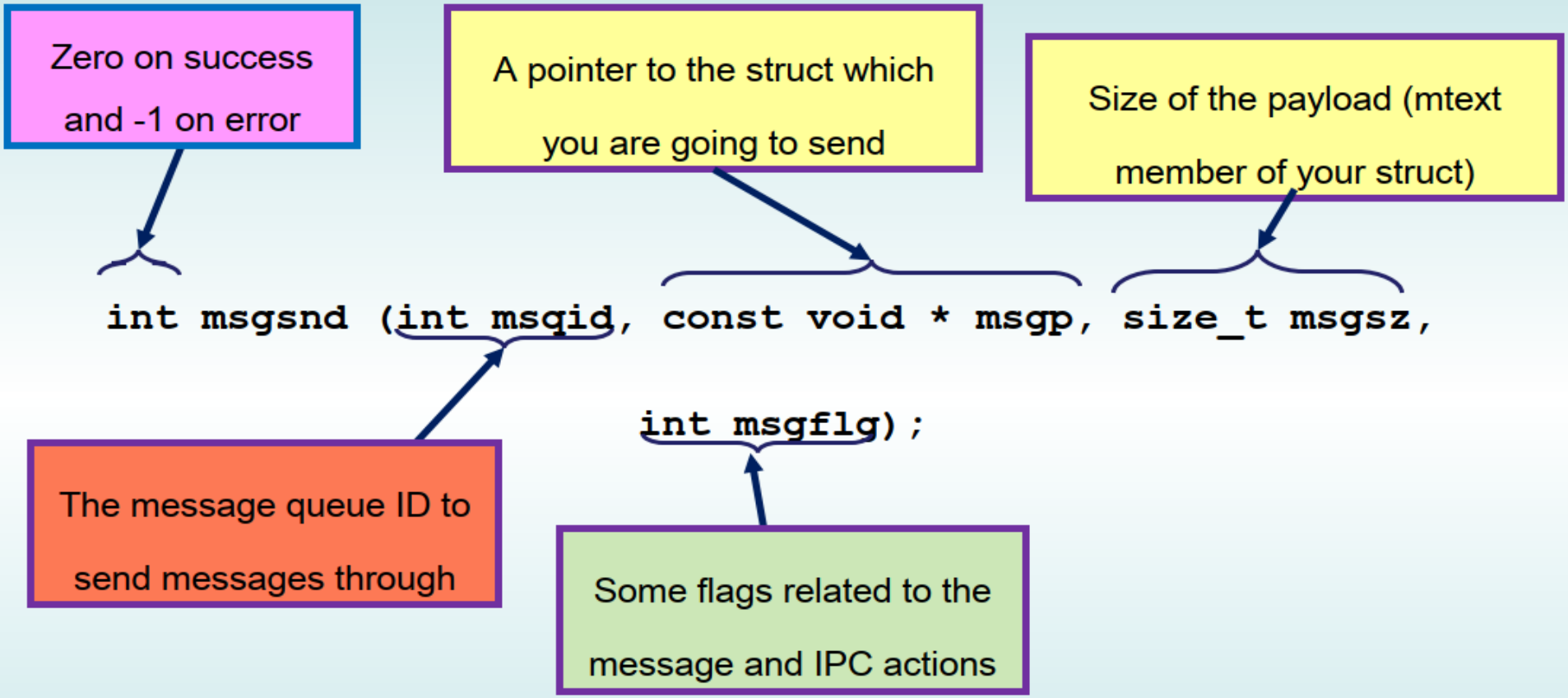
- You can use *ipcs* command to view the current message queues.

Message Queue



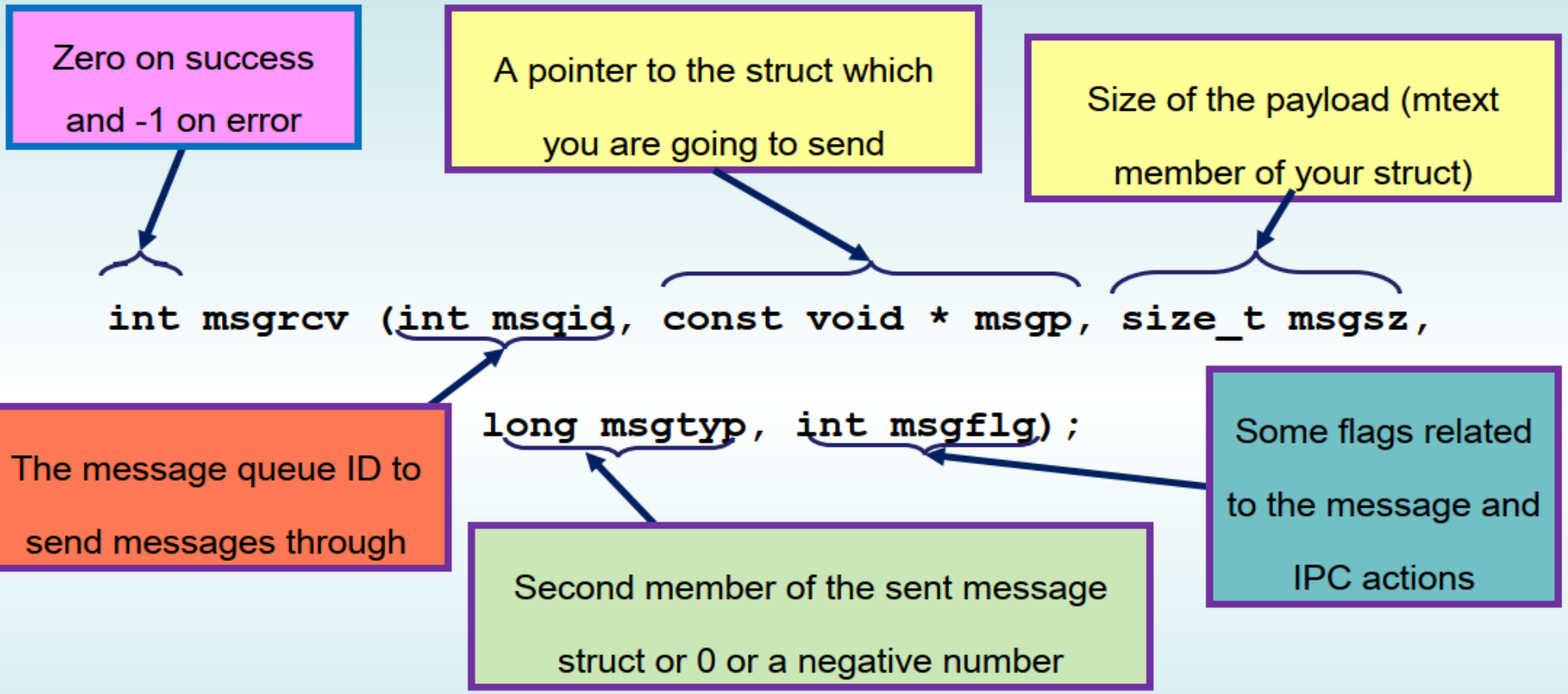
- You can connect to a queue and also create one, using *msgget()*.

Message Queue



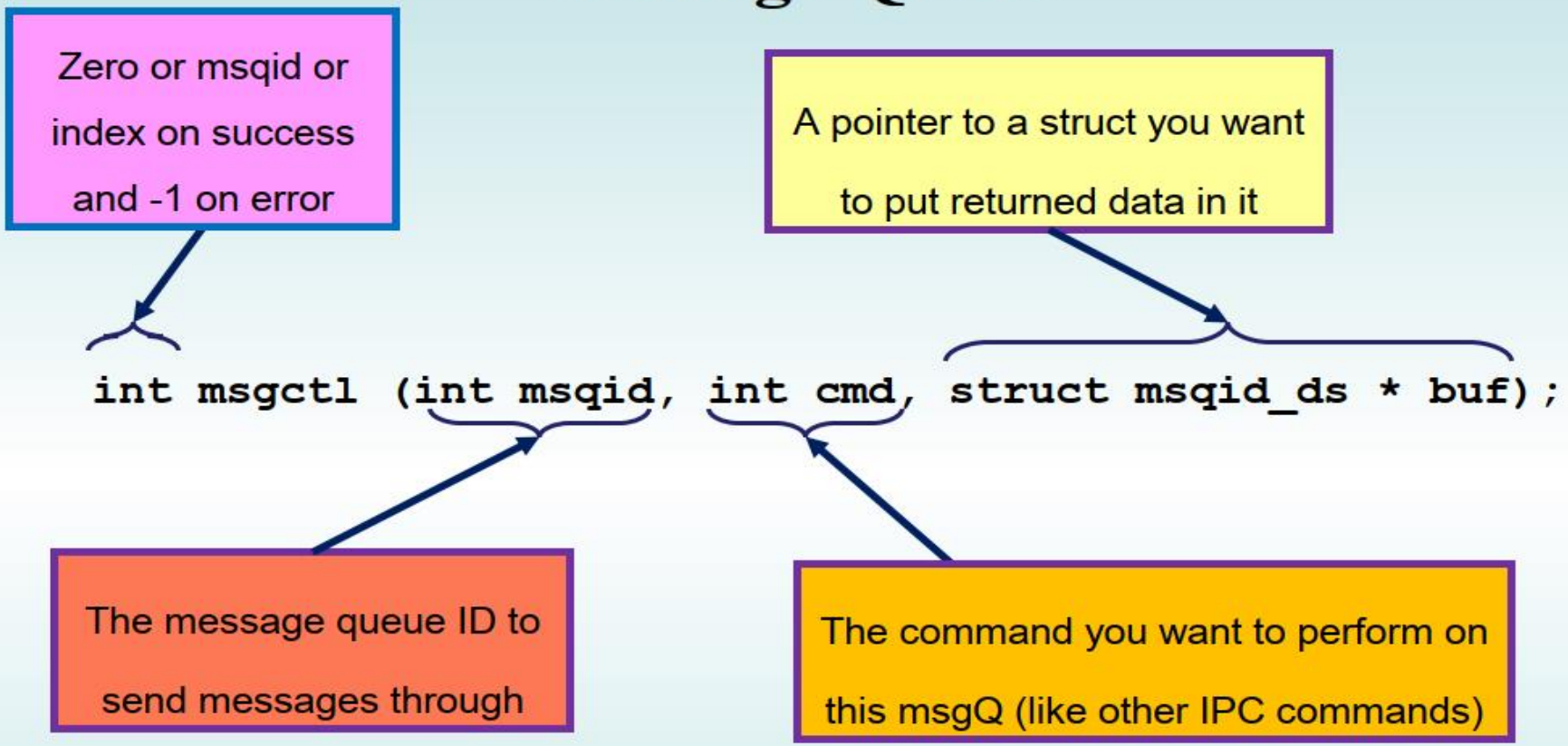
- The message you are going to send should be a struct containing two members: m_type and m_text.

Message Queue



- If the message has size bigger than what is said here, the truncated message could be lost (MSG_NOERROR flag)

Message Queue



- The msqid_ds type is described in sys/msg.h and contains information about desired message queue.

POSIX Message Queues

POSIX supports asynchronous, indirect message passing through the notion of message queues

A message queue can have many readers and many writers

Priority may be associated with the queue

Intended for communication between processes (not threads)

Message queues have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.

An attribute object is used to set the queue attributes when the queue is created

POSIX Message Queues

Message queues are given a name when they are created

To gain access to the queue, requires an `mq_open` `name`

`mq_open` is used to both create and open an already existing queue (also `mq_close` and `mq_unlink`)

Sending and receiving messages is done via `mq_send` and `mq_receive`

Data is read/written from/to a character buffer.

If the buffer is full or empty, the sending/receiving process is blocked unless the attribute O_NONBLOCK has been set for the queue (in which case an error return is given)

If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified

POSIX Message Queues

A process can also indicate that a signal should be sent to it when an empty queue receives a message and there are no waiting receivers

In this way, a process can continue executing whilst waiting for messages to arrive or one or more message queues

It is also possible for a process to wait for a signal to arrive; this allows the equivalent of selective waiting to be implemented

If the process is multi-threaded, each thread is considered to be a potential sender/receiver in its own right

Shared Memory

- Is the fastest form of interprocess communication and is also called: Fast Local Communication.

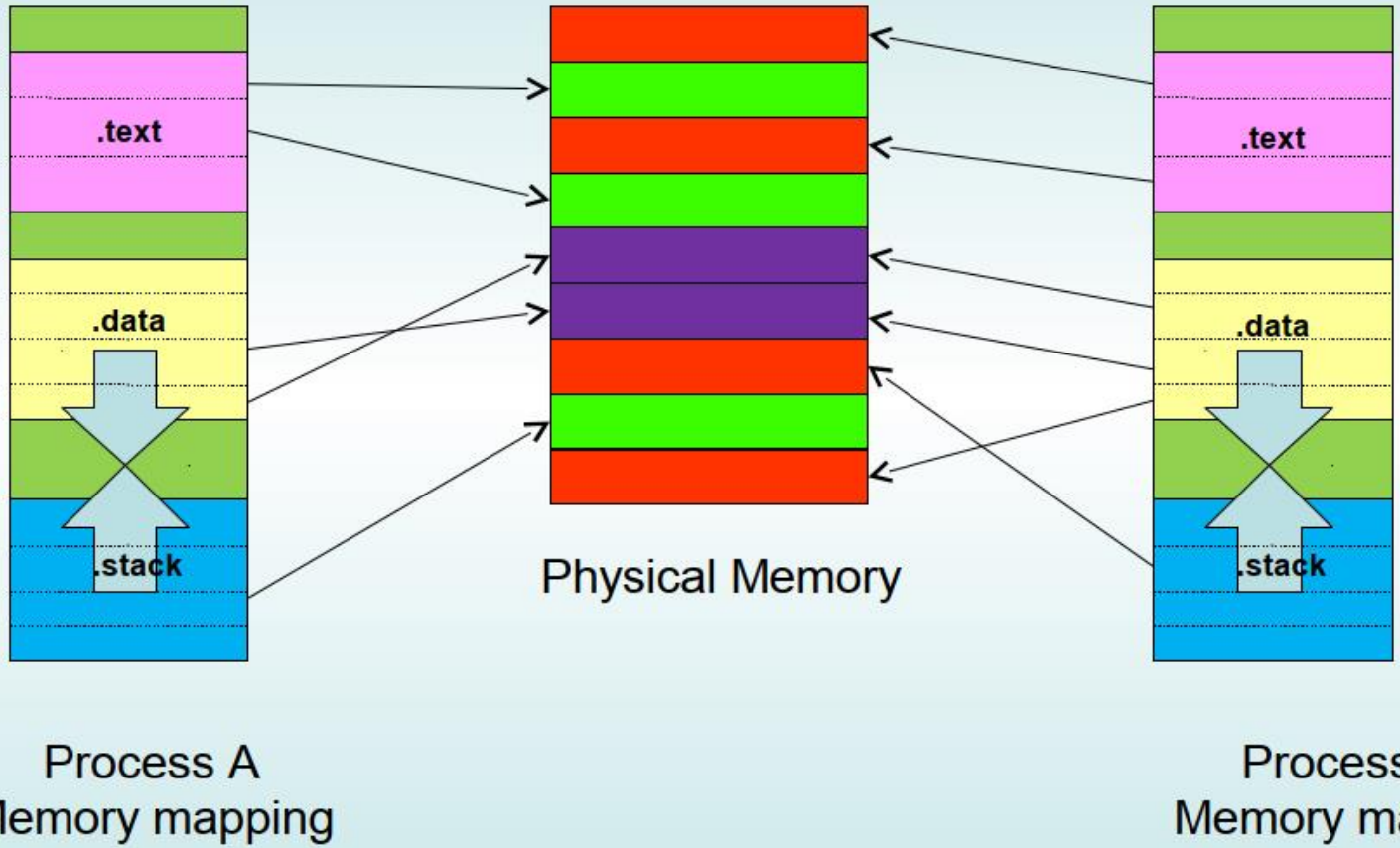
- It allows two or more processes to access the same memory.

- Linux kernel does not take care of synchronization between processes to access the shared memory.

- Process semaphores are a suitable way of synchronization.

- Using shared memory for each process is like calling *malloc.*

Shared Memory



Using Shared Memory

- To use shared memory in Linux, one process must allocate the segment.

- Each process desiring to access the segment, must attach the segment and after finishing its job must detach the segment.

- One process at the end must deallocate the segment.

- Allocating a new shared memory segment, causes virtual memory pages to be created.

Using Shared Memory

- Allocating an existing segment, does not create new pages, but returns and identifier to the existing ones.

- All shared memory segments are allocated as integral multiples of the system's page size.

- On Linux systems, page size is 4KB.

- You should obtain the page size by calling the *getpagesize()* function.

Using Shared Memory

- You can use *ipcs -m* command to see currently assigned shared segments.

- Using *ipcrm -m* command, you can remove unused shared segments left behind by processes.

- There are some limitations on using shared memory in linux.

- *SHMALL, SHMMAX, SHMMIN, SHMMNI* are corresponding values to these limitations which are located under /proc/sys/kernel/.

Using Shared Memory

- SHMALL: system-wide maximum of shared memory pages.

- SHMMAX: maximum size in bytes for a shared memory segment.

- SHMMIN: minimum size in bytes for a shared memory segment.

- SHMMNI: system-wide maximum number of shared segments.

If the minimum size of a shared memory segment is equal to *page size*, then what is *SHMMIN*?

Using Shared Memory

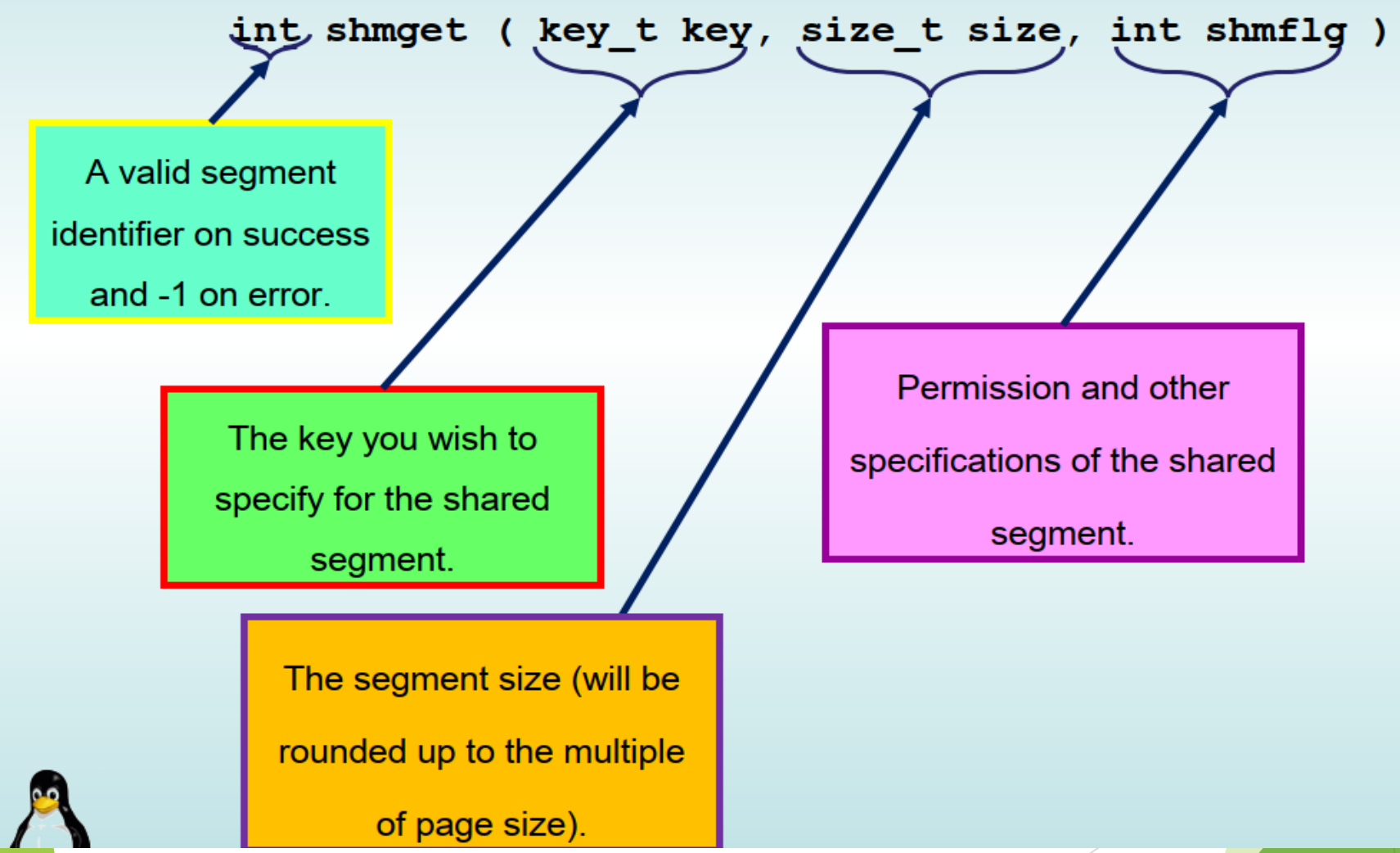
- A process may allocate a shared memory segment using *shmget()*.

- Its first argument is a key specified to the shared segment.

- Other processes can access to the same shared memory segment using the same key.

- To ensure that the key is not previously used, you can use the special constant IPC_PRIVATE.

Using Shared Memory



Using Shared Memory

- shmflg is the logical OR of flags. The most useful flags are:

  • IPC_CREAT: Is used to create a new shared segment.

  • IPC_EXCL: Is used with IPC_CREAT to ensure failure if the segment already exists.

  • Mode flags (see the manual page of stat.h for details).

If IPC_CREAT is used without IPC_EXCL, and the segment key already exists, the existing segment's id will be returned and no error will occur.

Using Shared Memory

- Permission flags are:

Mode bit	Meaning
S_IRWXU	R, W, X by owner
S_IRUSR	Read by owner
S_IWUSR	Write by owner
S_IXUSR	Execute by owner
S_IRWXG	R, W, X by group
S_IRGRP	Read by group
S_IWGRP	Write by group
S_IXGRP	Execute by group
S_IRWXO	R, W, X by other
S_IROTH	Read by other
S_IWOTH	Write by other
S_IXOTH	Execute by other

Using Shared Memory

- To use a shared memory segment, a process must attach it.

- *shmat()* is used to attach to a shared memory segment with given segment identifier.

- You can tell *shmat()* where in your process address space to map the shared memory.

- If you call *fork()* the child will inherit the shared memory.

- When you finished with shared memory, you can detach it using *shmdt()*.

Using Shared Memory

```
void * shmat ( int shmid, const void * shmaddr, int
shmflg )
```

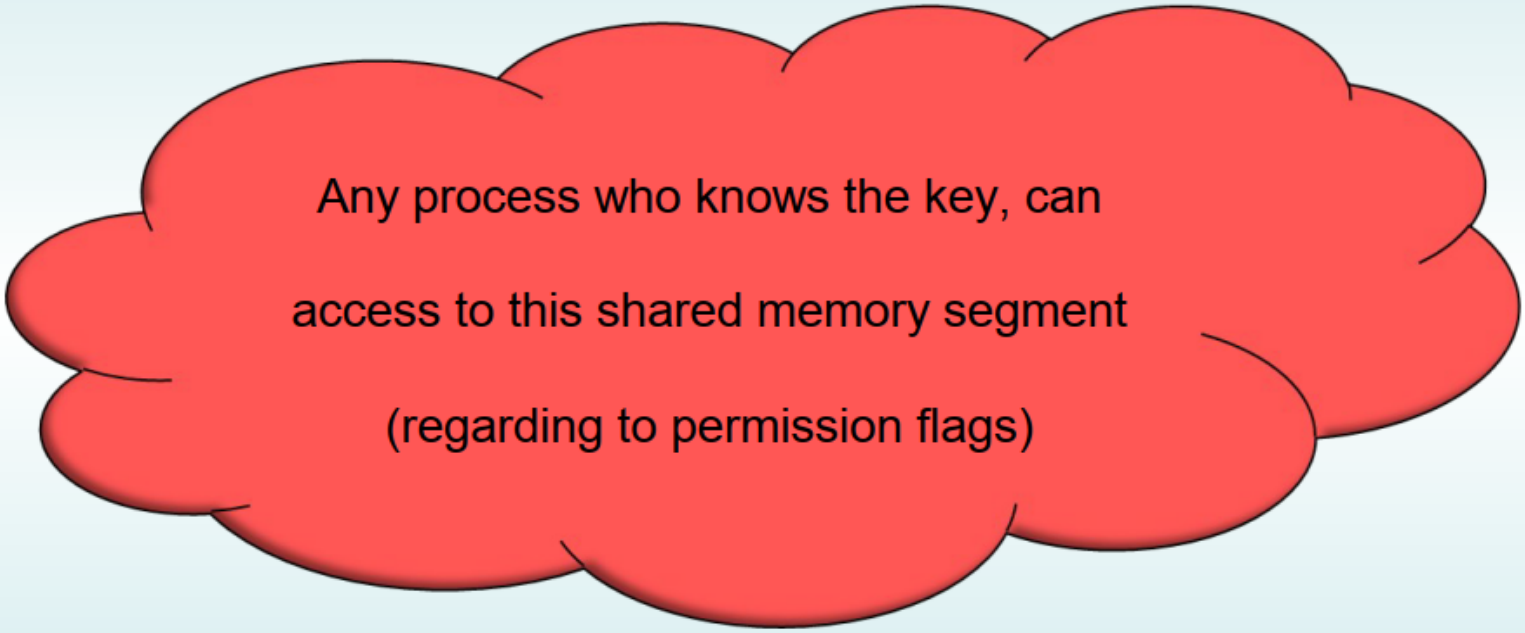
On success, will return the address of attached shared memory. On error, -1 is returned and errno is set.

The address in your process address space in which you want the shared memory be mapped.

The segment ID (returned by *shmget()*).

Could be ***SHM_RND,*** ***SHM_RDONLY,*** ***SHM_REMAP*** *(Linux specific)*

Using Shared Memory



Using Shared Memory

- The *shmctl()* returns information about a shared memory segment and can modify it.

- Using *shmctl()* you can also deallocate a shared memory segment.

- Each shared memory segment should be deallocated explicitly.

- The *shmctl()* fills the *shmid_ds* type structure.

Using Shared Memory

```
int shmctl ( int id, int cmd, struct shmid_ds * buf )
```

On success depends on cmd. On error, -1 is returned and errno is set.

The segment ID (returned by *shmget()*).

IPC_STAT, IPC_SET, IPC_RMID, RPC_INFO *(Linux specific),* ***SHM_INFO*** *(Linux specific),* ***SHM_STAT*** *(Linux specific),* ***SHM_LOCK*** *(Linux specific),* ***SHM_UNLOCK*** *(Linux specific).*

An structure which contains the information you want to be set or to be read.

Process Semaphore

- Processes must coordinate access to shared memory.

- Process semaphores like thread semaphores are kind of counter with two operations: *POST* & *WAIT*.

- Process semaphores come in sets.

- The last process using a semaphore set, must explicitly remove it.

- Unlike shared memory, removing a semaphore set causes Linux to deallocate immediately.

Process Semaphore

- To use a semaphore set, you should first allocate it using *semget()*.

- The *semget()* system call, will return a semaphore ID regarding to the key you gave it before.

- After allocating the semaphore, you should initialize it using *semctl()* .

- After initializing the semaphore set, you can do POST or WAIT on it using *semop()* system call.

- The last process must invoke *semctl()* to remove the semaphore.

Process Semaphore

- Allocating an existing semaphore will return it's semaphore ID.

- *semget()* flags behave the same way as *shmget()* does.

- You can use *ipcs –s* command to view information about existing semaphore sets.

- Using *ipcrm –s* you may remove the semaphore sets.

- Each semaphore in a set has the following associated values:

```
unsigned short semval;      /* semaphore value */
unsigned short semzcnt;     /* # waiting for zero */
unsigned short semncnt;     /* # waiting for increase */
pid_t sempid;               /* PID that did last OP */
```


Allocating a semaphore set

```
int semget ( key_t key, int nsems, int semflg )
```

A valid semaphore identifier on success and -1 on error.

The key you wish to specify for the semaphore set.

The number of semaphores you wish to have in this set.

Permission and other specifications of the semaphore set.

Initializing Semaphores

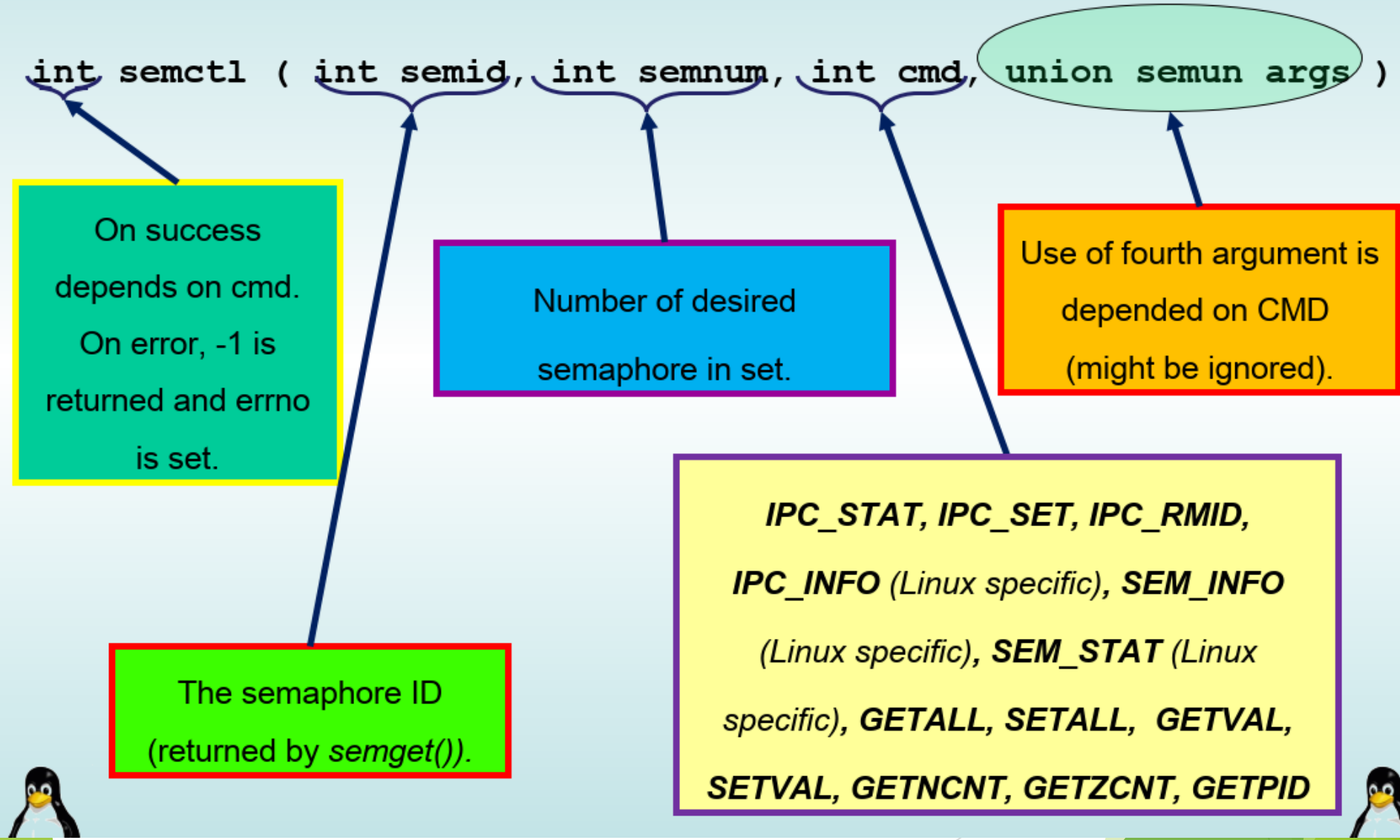
- To initialize a semaphore set, you must do:

  - Set the semaphore value of all members to desired values.
  - Set the last change time of all members.
  - Set other specifications of semaphore set members.

- To do so, you can use *semctl()* function.

- As mentioned in *semctl()* manual page, the calling program must define a *semun* union.

Initializing Semaphores



Initializing Semaphores

- Depending on CMD, you may need to provide the fourth argument.

- If so, you must define the *union semun* yourself like below:

```
union semun
{
        int val;    /* Value for SETVAL */
        struct semid_ds * buf;    /* Buffer for IPC_STAT, IPC_SET */
        unsigned short int * array;  /* Array for GETALL, SETALL */
        struct seminfo * __buf; /* Buffer for IPC_INFO (linux specific) */
};
```

semid_ds and *ipc_perm* (an structure in *semid_ds*) are filled with information about semaphores

Wait and post operation

- To do *wait* and *post* on a semaphore in a set, you can use *semop()*.

- *semop()* performs desired operation on selected semaphores in a set.

- *semop()* takes an array of operation structure.

- Each operation structure is related to one semaphore in a set.

- Operation structure is of type *sembuf* and contains:

```
unsigned short sem_num;   /* Semaphore number */
short sem_op;    /* Semaphore operation */
short sem_flg;    /* Semaphore flags */
```

Semaphore Deallocation

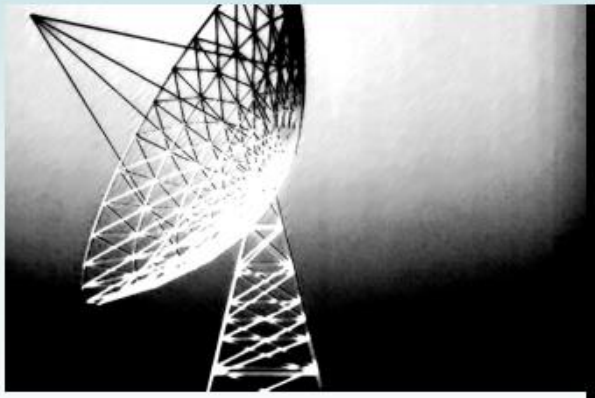
- The semaphore operations are performed atomically.

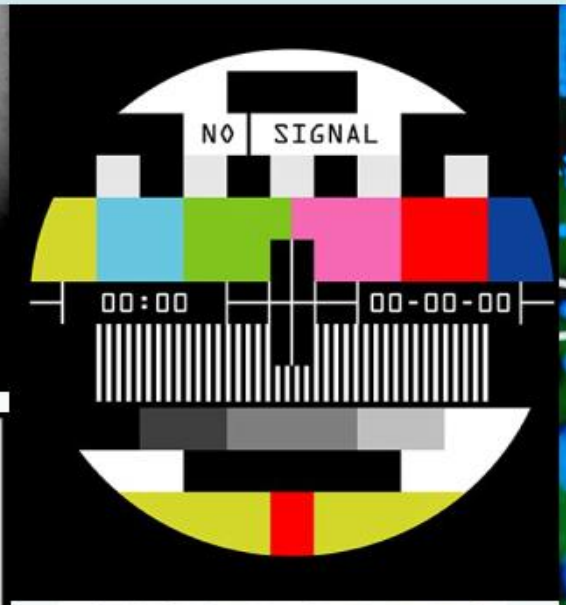
- If the *SEM_UNDO* flag is set in operation structure, the action will be undone when process terminates.

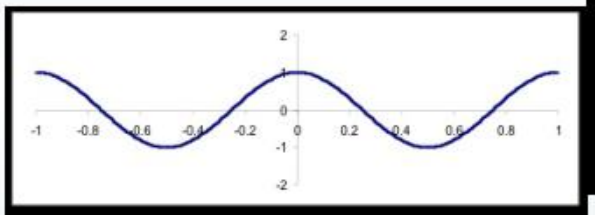
- You must deallocate the semaphore set when you finnished.

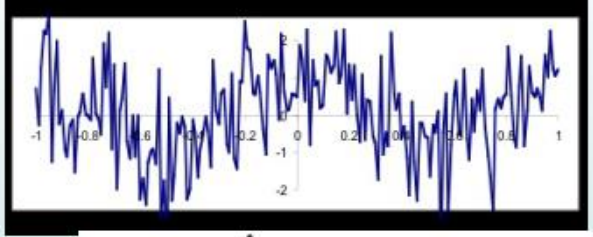
- Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.

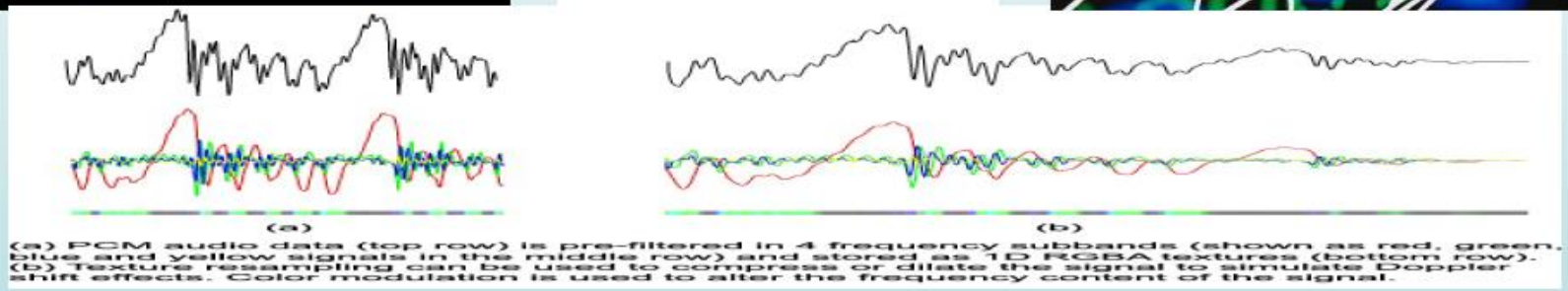
Signals











(a) PCM audio data (top row) is pre-filtered in 4 frequency subbands (shown as red, green, blue and yellow signals in the middle row) and stored as 1D RGBA textures (bottom row). (b) Texture resampling can be used to compress or dilate the signal to simulate Doppler shift effects. Color modulation is used to alter the frequency content of the signal.

Signals

- Signals are mechanisms for communicating with and manipulating processes in Linux.

- Signals are asynchronous software interrupts.

- In Linux, each signal has it's specific number.

- Signal names and numbers are defined in "/usr/include/bits/signum.h"

- A program may receive signals from *OS Itself, Other processes or users.*

```
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/IPCSemaphore$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/IPCSemaphore$
```

Signals

- A program may do one of lots of things when it receives a signal.

- For each signal there is a *Default Disposition* which determines the default behavior of a program when it receives this signal (If the program does not specify some specific action).

- There are some ways to handle the signals in Linux.

-The *SIGACTION* function can be used to change the action taken by a process on receipt of a specific signal.

- A signal is an *asynchronous* event which is delivered to a process.

- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types `ctrl-C`, or the modem hangs

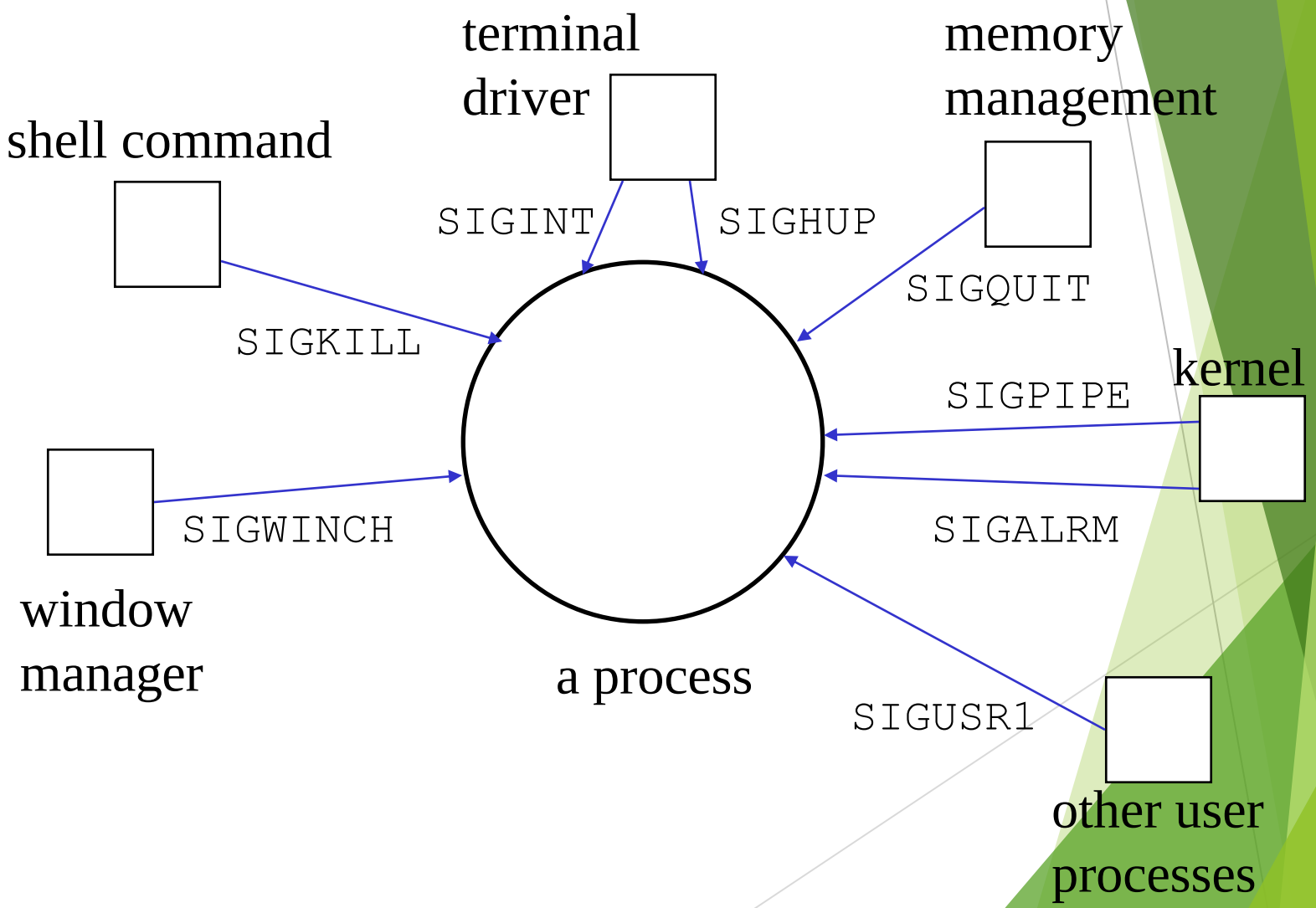
2. Signal Types (31 in POSIX)

- *Name Description Default Action*

  `SIGINT` Interrupt character typed terminate process
  `SIGQUIT` Quit character typed (`^\`) create core image
  `SIGKILL` `kill -9` terminate process
  `SIGSEGV` Invalid memory reference create core image
  `SIGPIPE` Write on pipe but no reader terminate process
  `SIGALRM` `alarm()` clock 'rings' terminate process
  `SIGUSR1` user-defined signal type terminate process
  `SIGUSR2` user-defined signal type terminate process

- See `man 7 signal`

Signal Sources



3. Generating a Signal

- Use the UNIX command:

```
$ kill -KILL 4481
```

send a `SIGKILL` signal to pid 4481

check

- **`ps -l`**

to make sure process died

- `kill` is not a good name; `send_signal` might be better.

kill()

- Send a signal to a process (or group of processes).
-
```
#include <signal.h>

int kill( pid_t pid, int
signo );
```

- Return 0 if ok, -1 on error.

Some pid Values

- *pid* *Meaning*

  > 0 send signal to process `pid`

  == 0 send signal to all processes whose process group ID equals the sender's pgid.

  e.g. parent kills all children

  - 1 send to all processes it can send signal to

  - n send to all processes with process group ID = n

4. Responding to a Signal

- A process can:

  ignore/discard the signal (not possible with `SIGKILL` or `SIGSTOP`)

  execute a **signal handler** function, and then possibly resume execution or terminate

  carry out the default action for that signal

- The choice is called the process' *signal disposition*

signal(): library call

- Specify a signal handler function to deal with a signal type.

- ```
  #include <signal.h>
  typedef void Sigfunc(int);  /* my defn */

  Sigfunc *signal( int signo, Sigfunc *handler );
  ```

  signal returns a pointer to a function that returns an int (i.e. it returns a pointer to Sigfunc)

- Returns *previous* signal disposition if ok, `SIG_ERR` on error.
  ```

Actual Prototype

The actual prototype, listed in the “man” page is a bit perplexing but is an expansion of the `Sigfunc` type:

```
void (*signal(int signo, void(*handler)(int)))(int);
```

In Linux:

```
typedef void (*sighandler_t)(int);

sig_handler_t signal(int signo, sighandler_t handler);
```

Signal returns a pointer to a function that returns an int

The signal function itself returns a pointer to a function. The return type is the same as the function that is passed in, i.e., a function that takes an int and returns a void

The *handler* function Receives a single integer Argument and returns *void*

- Then can use the *sig*

```
#include <signal.h>

void (*signal( int sig, void (*handler)(int))) (int) ;
```

- signal returns a poin

```
.h>
rfunc(int);
Sigfunc *signal( int signo, Sigfunc *handler );
```

Signal is a function that takes two arguments: *sig* and *handler*

The function to be called when the specified signal is receiv pointer t *handler*

The returned function takes a integer parameter.

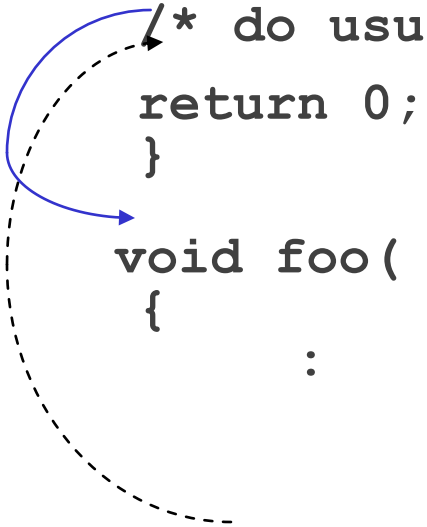
Example

```
int main()
 {
    signal( SIGINT, foo );
      :

 /* do usual things until SIGINT */
 return 0;
 }

void foo( int signo )
 {
      :       /* deal with SIGINT signal */


    return;     /* return to program */
 }
```



sig_examp.c

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void sig_usr( int signo );     /* handles two signals */


int main()
   {
   int i = 0;
   if( signal( SIGUSR1,sig_usr ) == SIG_ERR )
    printf( "Cannot catch SIGUSR1\n" );
   if( signal( SIGUSR2,sig_usr ) == SIG_ERR )
      printf("Cannot catch SIGUSR2\n");
          :
```

continued

```
    :
   while(1)
    {
       printf( "%2d\n", I );
       pause();
       /* pause until signal handler
           * has processed signal */
       i++;
       }
   return 0;
}
```

continued

```
void sig_usr( int signo )
/* argument is signal number */
{
  if( signo == SIGUSR1 )
     printf("Received SIGUSR1\n");
  else if( signo == SIGUSR2 )
     printf("Received SIGUSR2\n");
  else
     printf("Error: received signal
        %d\n", signo);

  return;
}
```

Usage

```
$ sig_examp &
[1]      4720
 0
$ kill -USR1 4720
Received SIGUSR1
 1
$ kill -USR2 4720
Received SIGUSR2
 2
$ kill 4720          /* send
SIGTERM */
[1] + Terminated    sig_examp &
$
```

Special Sigfunc * Values

Value	*Meaning*
`SIG_IGN`	Ignore / discard the signal.
`SIG_DFL`	Use default action to handle signal.
`SIG_ERR`	Returned by `signal()` as an error.

Multiple Signals

- If many signals of the *same* type are waiting to be handled (e.g. two `SIGINT`s), then most UNIXs will only deliver one of them.

  the others are thrown away

- If many signals of *different* types are waiting to be handled (e.g. a `SIGINT`, `SIGSEGV`, `SIGUSR1`), they are not delivered in any fixed order.

pause()

- Suspend the calling process until a signal is caught.
-
```
#include <unistd.h>
int pause(void);
```
- Returns -1 with `errno` assigned `EINTR`. (Linux assigns it `ERESTARTNOHAND`).
- `pause()` only returns after a signal handler has returned.

The Reset Problem

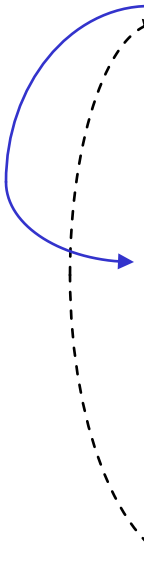
- In Linux (and many other UNIXs), the signal disposition in a process is **reset** to its **default action** immediately after the signal has been delivered.

- Must call `signal()` again to reinstall the signal handler function.

Reset Problem Example

```
int main()
 {
    signal(SIGINT, foo);
     :
    /* do usual things until SIGINT */
 }

void foo(int signo)
 {
    signal(SIGINT, foo); /* reinstall */
    :
    return;
 }
```



Reset Problem

```
      :
void ouch( int sig )
    {
       printf( "OUCH! - I got signal %d\n", sig );
    (void) signal(SIGINT, ouch
    }
int main()
    {
       (void) signal( SIGINT,
       while(1)
       {
          printf("Hello World!\n");
          sleep(1);
```

To keep catching the signal with this function, must call the *signal* system call again.

Problem: from the time that the interrupt function starts to just before the signal handler is re-established the signal will not be handled.

If another SIGINT signal is received during this time, default behavior will be done, i.e., program will terminate.

Re-installation may be too slow!

- There is a (very) small time period in `foo()` when a new `SIGINT` signal will cause the default action to be carried out -- process termination.

- With `signal()` there is no answer to this problem.

  POSIX signal functions solve it (and some other later UNIXs)

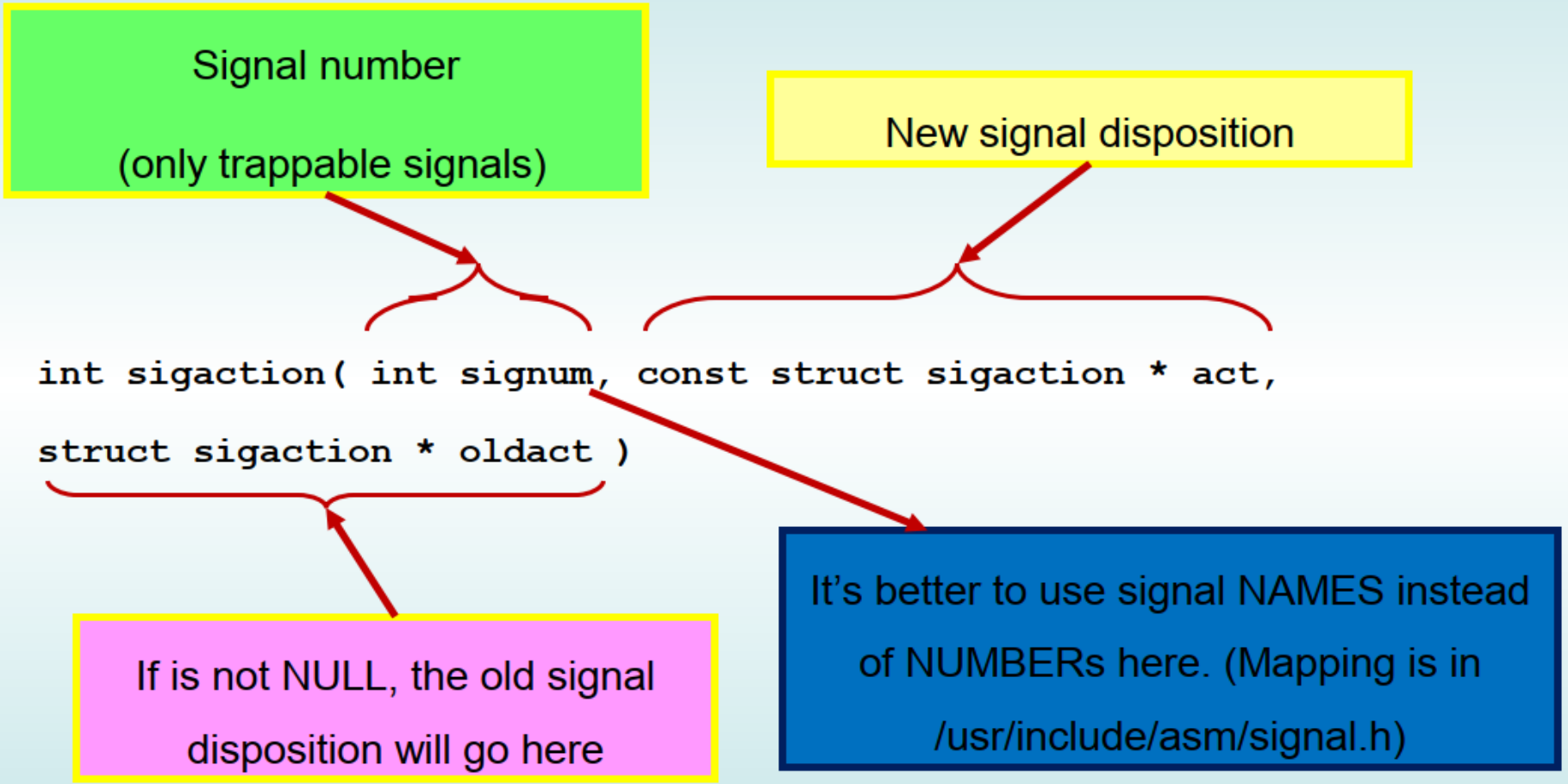
alarm()

- Set an alarm timer that will ‘ring’ after a specified number of seconds
  - a `SIGALRM` signal is generated

```
#include <unistd.h>
long alarm(long secs);
```

- Returns 0 or number of seconds until previously set alarm would have ‘rung’.

Signals


Signal number
(only trappable signals)
New signal disposition
int sigaction(int signum, const struct sigaction * act,
struct sigaction * oldact)
If is not NULL, the old signal disposition will go here
It's better to use signal NAMES instead of NUMBERs here. (Mapping is in /usr/include/asm/signal.h)

Signals

- The sigaction structure is something like this:

SIG_IGN, SIG_DFL or a handler function which takes an integer as the signal number

```
struct sigaction
{
        void (*sa_handler) (int);
        void (*sa_sigaction) (int, siginfo_t *, void *);
        sigset_t sa_mask;
        int sa_flags;
        void (* sa_restorer) (void);
}
```

If you want more information about the received signal, you can set this function instead of *sa_handler*.