

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

A screenshot of a Linux desktop environment. At the top, there is a docked application bar with icons for various applications: a browser (Firefox), file manager (Nautilus), terminal (Konsole), file manager (Thunar), terminal (Konsole), file manager (Thunar), terminal (Konsole), and help (Help). Below the dock, a terminal window is open with the following command-line session:

```
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -o hello hello.s
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./hello
Hello World
test@test-VirtualBox:/media/sf_shared/AdvancedC$
```

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; 1 st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4 th integer argument to functions	No
%rdx	used to pass 3 rd argument to functions; 2 nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2 nd argument to functions	No
%rdi	used to pass 1 st argument to functions	No
%r8	used to pass 5 th argument to functions	No
%r9	used to pass 6 th 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.
#
# -----
```

global add

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

X86 Assembly

```
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -o add_main add_main.c add.s
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./add_main
3
-4
5
2
-4
6
test@test-VirtualBox:/media/sf_shared/AdvancedC$
```

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

test@test-VirtualBox: /media/sf_shared/AdvancedC\$

```
test@test-VirtualBox: /media/sf_shared/AdvancedC$ gcc -g -o inline inline.c
test@test-VirtualBox: /media/sf_shared/AdvancedC$ ./inline
foo+bar=25
test@test-VirtualBox: /media/sf_shared/AdvancedC$
```



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

test@test-VirtualBox:/media/sf_shared/AdvancedC\$

```
gcc -c -o obj1.o obj1.c
gcc -c -o obj2.o obj2.c
ar cr libfuncs.a obj1.o obj2.o
file libfuncs.a
libfuncs.a: current ar archive
```

test@test-VirtualBox:/media/sf_shared/AdvancedC\$



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

```
test@test-VirtualBox:/media/sf_shared/Advanced$ gcc -L . -lfuncs -o statictest statictest.c  
/usr/bin/ld: /tmp/ccb8NYnK.o: in function `main':  
statictest.c:(.text+0x9): undefined reference to `my_func1'  
/usr/bin/ld: statictest.c:(.text+0xe): undefined reference to `my_func2'  
collect2: error: ld returned 1 exit status  
test@test-VirtualBox:/media/sf_shared/Advanced$  
test@test-VirtualBox:/media/sf_shared/Advanced$  
test@test-VirtualBox:/media/sf_shared/Advanced$  
test@test-VirtualBox:/media/sf_shared/Advanced$  
test@test-VirtualBox:/media/sf_shared/Advanced$
```

Static Library

```
test@test-VirtualBox:/media/sf_shared/Advanced$ gcc -L. -lfuncs -o statictest statictest.c
/usr/bin/ld: /tmp/ccb8NYnK.o: in function `main':
statictest.c:(.text+0x9): undefined reference to `my_func1'
/usr/bin/ld: statictest.c:(.text+0xe): undefined reference to `my_func2'
collect2: error: ld returned 1 exit status
test@test-VirtualBox:/media/sf_shared/Advanced$ 
test@test-VirtualBox:/media/sf_shared/Advanced$ 
test@test-VirtualBox:/media/sf_shared/Advanced$ 
test@test-VirtualBox:/media/sf_shared/Advanced$ 
test@test-VirtualBox:/media/sf_shared/Advanced$ gcc -L. -o statictest statictest.c -lfuncs
test@test-VirtualBox:/media/sf_shared/Advanced$ 
test@test-VirtualBox:/media/sf_shared/Advanced$ ./statictest
This is func1
This is func2
test@test-VirtualBox:/media/sf_shared/Advanced$ 
```

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

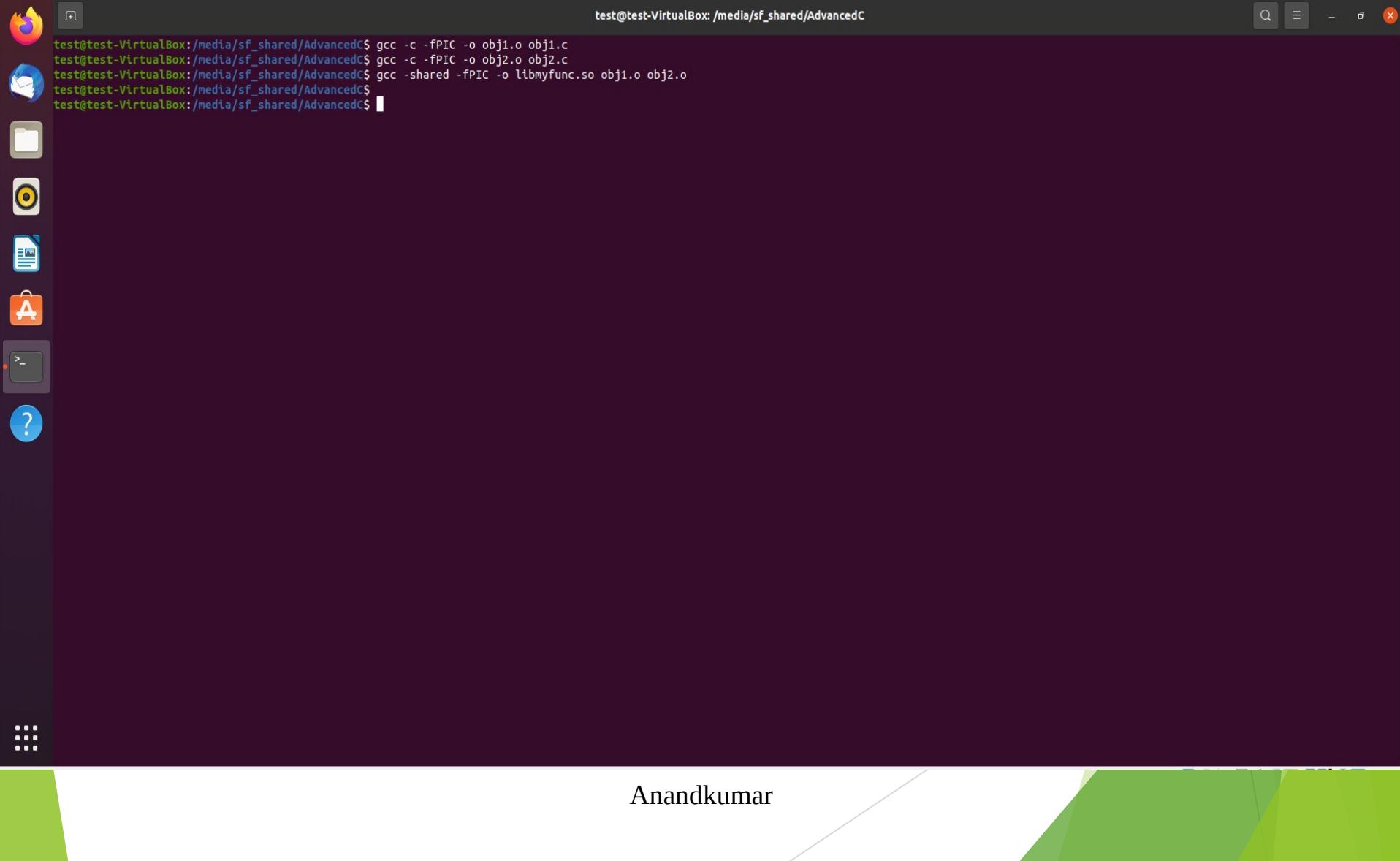
- The linker just includes the name of the “so” in executable file.
- The Operating System is responsible to find the specified “so” file.
- By default, system searches only “/lib” and “/usr/lib”.
- You can indicate another path by setting the `LD_LIBRARY_PATH` environment variable.



Shared Library

test@test-VirtualBox: /media/sf_shared/AdvancedC\$

```
gcc -c -fPIC -o obj1.o obj1.c
gcc -c -fPIC -o obj2.o obj2.c
gcc -shared -fPIC -o libmyfunc.so obj1.o obj2.o
test@test-VirtualBox: /media/sf_shared/AdvancedC$
```



Shared Library

```
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -c -fPIC -o obj1.o obj1.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -c -fPIC -o obj2.o obj2.c
test@test-VirtualBox:/media/sf_shared/AdvancedC$ gcc -shared -fPIC -o libmyfunc.so obj1.o obj2.o
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 -o dynamictest dynamictest.c -L. -lmyfunc
test@test-VirtualBox:/media/sf_shared/AdvancedC$ 
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./dynamictest
./dynamictest: error while loading shared libraries: libmyfunc.so: cannot open shared object file: No such file or directory
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$ 
```

Shared Library

```
test@test-VirtualBox:/media/sf_shared/Advanced$ gcc -c -fPIC -o obj1.o obj1.c
test@test-VirtualBox:/media/sf_shared/Advanced$ gcc -c -fPIC -o obj2.o obj2.c
test@test-VirtualBox:/media/sf_shared/Advanced$ gcc -shared -fPIC -o libmyfunc.so obj1.o obj2.o
test@test-VirtualBox:/media/sf_shared/Advanced$ ./dynamicitest
./dynamicitest: error while loading shared libraries: libmyfunc.so: cannot open shared object file: No such file or directory
test@test-VirtualBox:/media/sf_shared/Advanced$ test@test-VirtualBox:/media/sf_shared/Advanced$ test@test-VirtualBox:/media/sf_shared/Advanced$ test@test-VirtualBox:/media/sf_shared/Advanced$ export LD_DEBUG=files
test@test-VirtualBox:/media/sf_shared/Advanced$ test@test-VirtualBox:/media/sf_shared/Advanced$ test@test-VirtualBox:/media/sf_shared/Advanced$ test@test-VirtualBox:/media/sf_shared/Advanced$ ./dynamicitest
25136:
25136:      file=libmyfunc.so [0];  needed by ./dynamicitest [0]
./dynamicitest: error while loading shared libraries: libmyfunc.so: cannot open shared object file: No such file or directory
test@test-VirtualBox:/media/sf_shared/Advanced$
```

Shared Library

```
test@test-VirtualBox:/media/sf_shared/AdvancedC$ ./dynamicctest
./dynamicctest: error while loading shared libraries: libmyfunc.so: cannot open shared object file: No such file or directory
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$ export LD_DEBUG=files
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$ ./dynamicctest
25136:
25136:     file=libmyfunc.so [0]; needed by ./dynamicctest [0]
./dynamicctest: error while loading shared libraries: libmyfunc.so: cannot open shared object file: No such file or directory
test@test-VirtualBox:/media/sf_shared/AdvancedC$ LD_LIBRARY_PATH=. ./dynamicctest
25138:
25138:     file=libmyfunc.so [0]; needed by ./dynamicctest [0]
25138:     file=libmyfunc.so [0]; generating link map
25138:         dynamic: 0x00007fc5983d0e20 base: 0x00007fc5983cd000 size: 0x0000000000004030
25138:             entry: 0x00007fc5983ce060 phdr: 0x00007fc5983cd040 phnum: 11
25138:
25138:
25138:     file=libc.so.6 [0]; needed by ./dynamicctest [0]
25138:     file=libc.so.6 [0]; generating link map
25138:         dynamic: 0x00007fc5983b2b80 base: 0x00007fc5981c8000 size: 0x00000000001f14d8
25138:             entry: 0x00007fc5981ef1f0 phdr: 0x00007fc5981c8040 phnum: 14
25138:
25138:
25138: calling init: /lib/x86_64-linux-gnu/libc.so.6
25138:
25138:
25138: calling init: ./libmyfunc.so
25138:
25138:
25138: initialize program: ./dynamicctest
25138:
25138:
25138: transferring control: ./dynamicctest
25138:
This is func1
This is func2
25138:
25138:     calling fini: ./dynamicctest [0]
25138:
25138:
25138:     calling fini: ./libmyfunc.so [0]
25138:
test@test-VirtualBox:/media/sf_shared/AdvancedC$
```

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

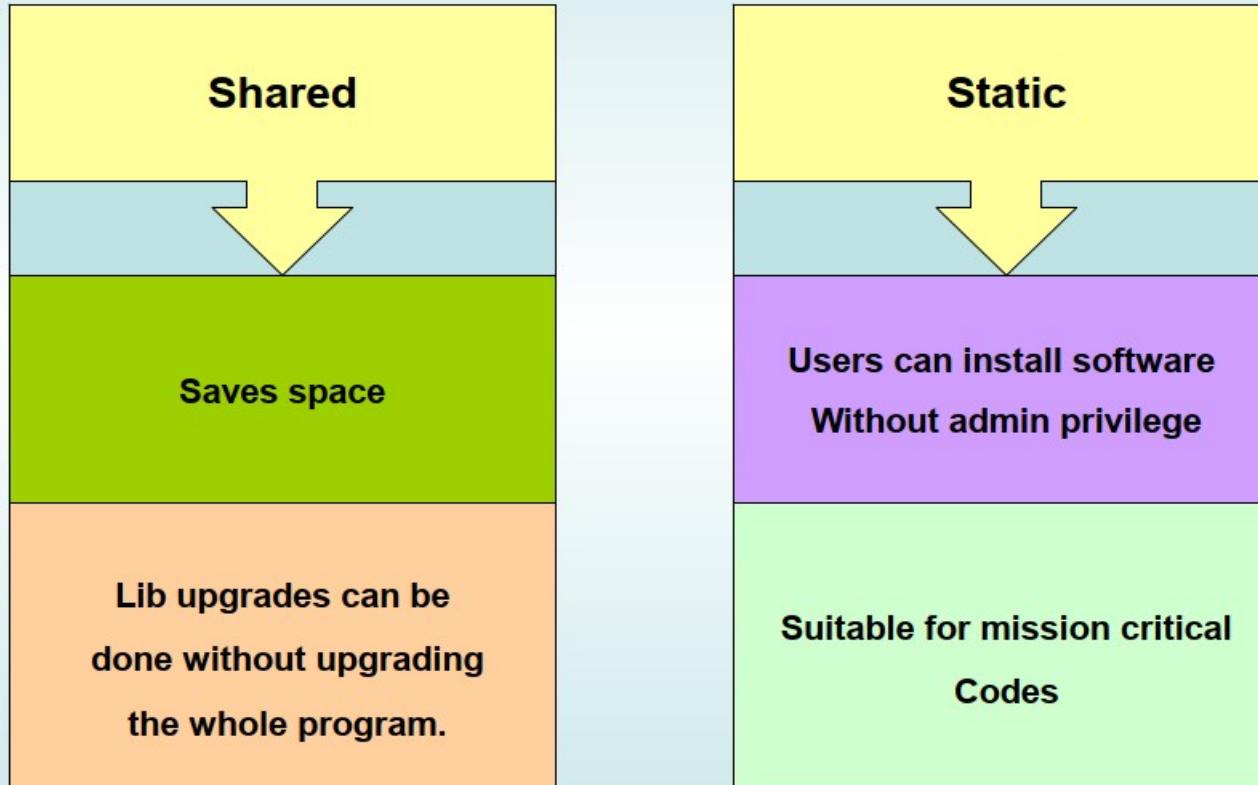
Shared Library

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.



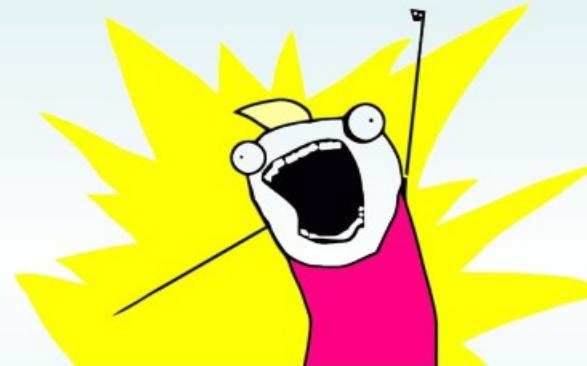
Shared vs. Static



Error Handling



CATCH ALL THE ERRORS!



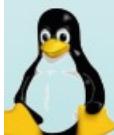
Error Handling

- 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$ ./handler  
Failed=13  
Permission denied  
test@test-VirtualBox: /media/sf_shared/AdvancedC$
```

Anandkumar

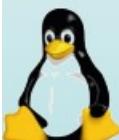
Table

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	EWOULD_BLOCK	(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$ 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$  
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;
}
```

A screenshot of a Linux desktop environment, likely Ubuntu, showing a terminal window in the foreground. The terminal window has a dark purple header bar with the title "test@test-VirtualBox: /media/sf_shared/AdvancedC". The main area of the terminal shows a command-line session where a C program is compiled and run. The session starts with several blank lines, followed by the command "gcc -g -o assert1 assert1.c", then the execution of the program "./assert1", which results in an assertion failure: "assert1: assert1.c:9: main: Assertion `i==2' failed. Aborted (core dumped)". This is followed by several blank lines, then the command "gcc -DNDEBUG -g -o assert1 assert1.c", the execution of the program "./assert1", and the message "The program ended successfully". The terminal window has a light gray background and a white cursor. The desktop environment includes a dock at the bottom with icons for Home, Dash, Activities, and Help, and a panel at the top with various system icons.

```
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$ 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

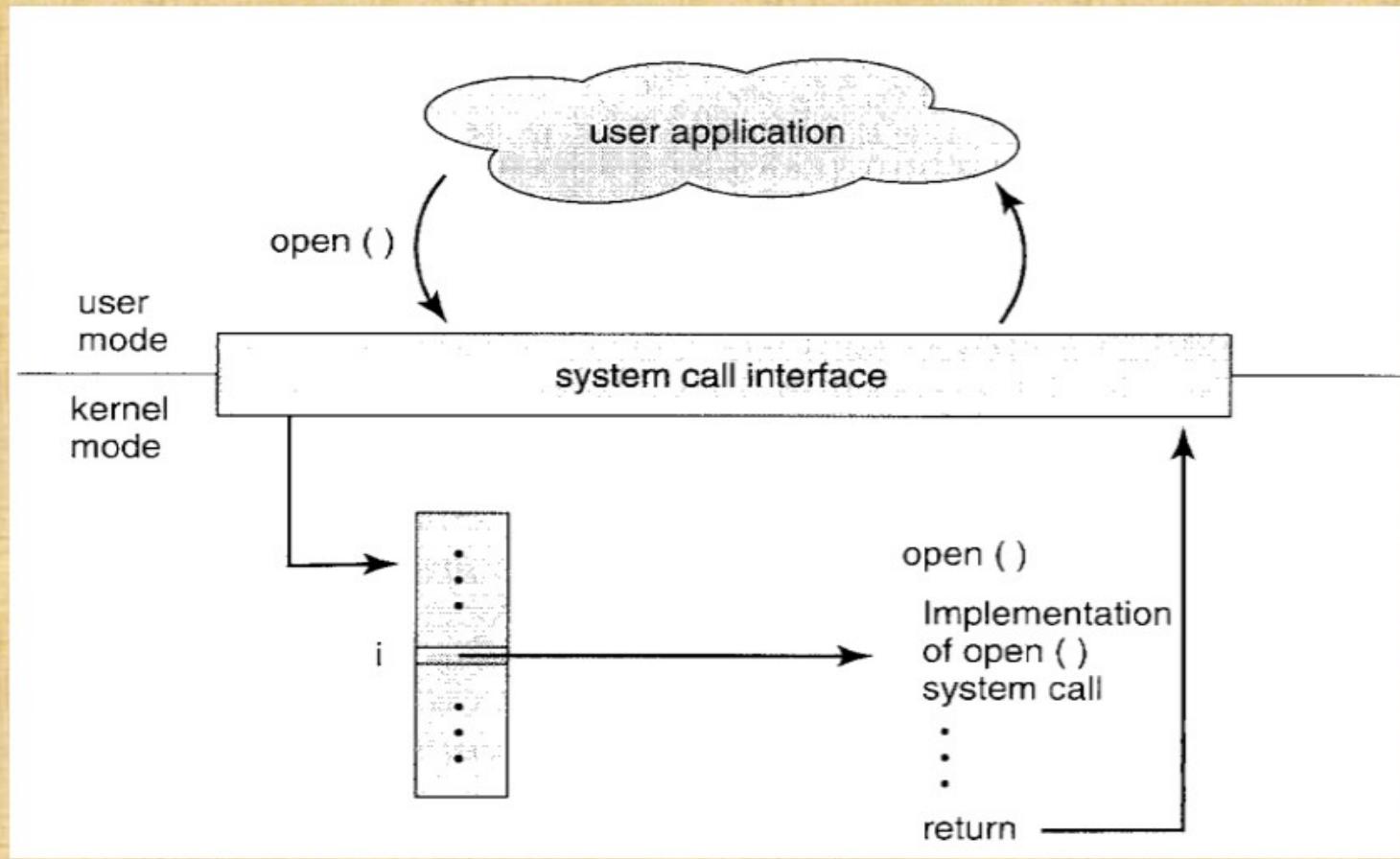
Given a pathname for a file, open() returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (read, write, lseek, fcntl, etc.).

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

1,9

All

Anandkumar



File

File contents
Test codetest@test-VirtualBox:/media/sf_shared/Advanced\$ cat testfile
Test code
testing_write_systemtest@test-VirtualBox:/media/sf_shared/Advanced\$
test@test-VirtualBox:/media/sf_shared/Advanced\$

|



test@test-VirtualBox:/media/sf_shared/Advanced\$

Q

≡

-

X

Anandkumar

Iseek function

Iseek - reposition read/write file offset

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>
off_t Iseek(int fd, off_t offset, int whence);
```

DESCRIPTION

The Iseek() 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, Iseek() 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$ 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$  
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);
```

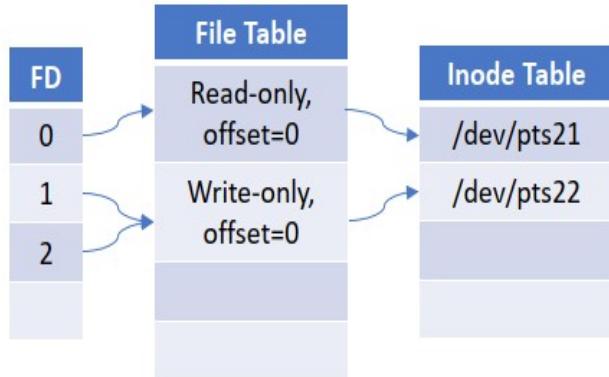
Both return: new file descriptor if OK,-1 on error.

- 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);`

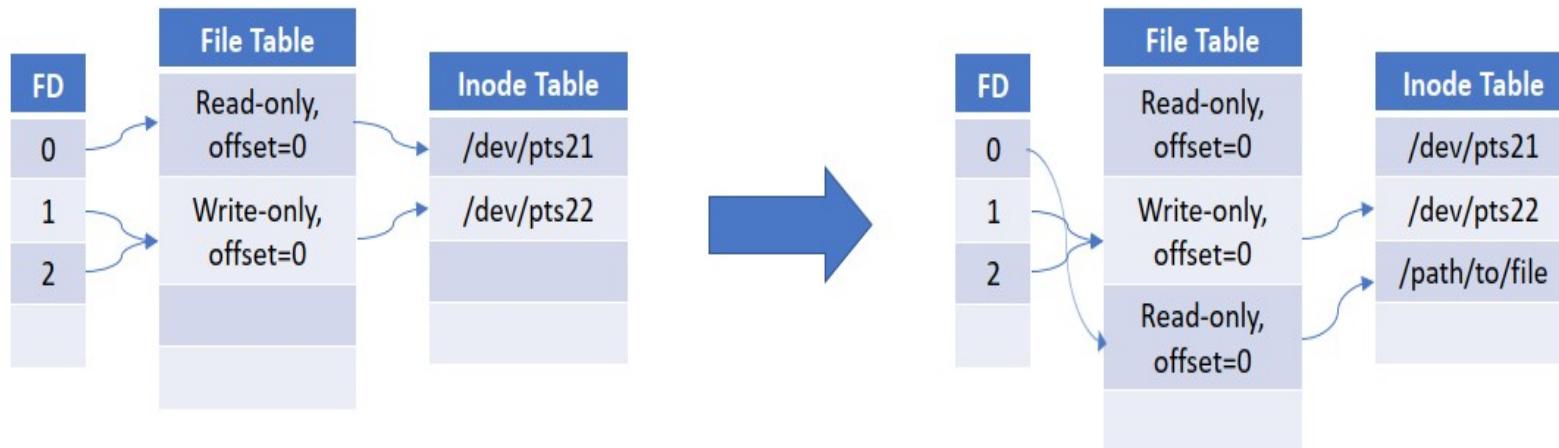
Example: Input Redirection

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



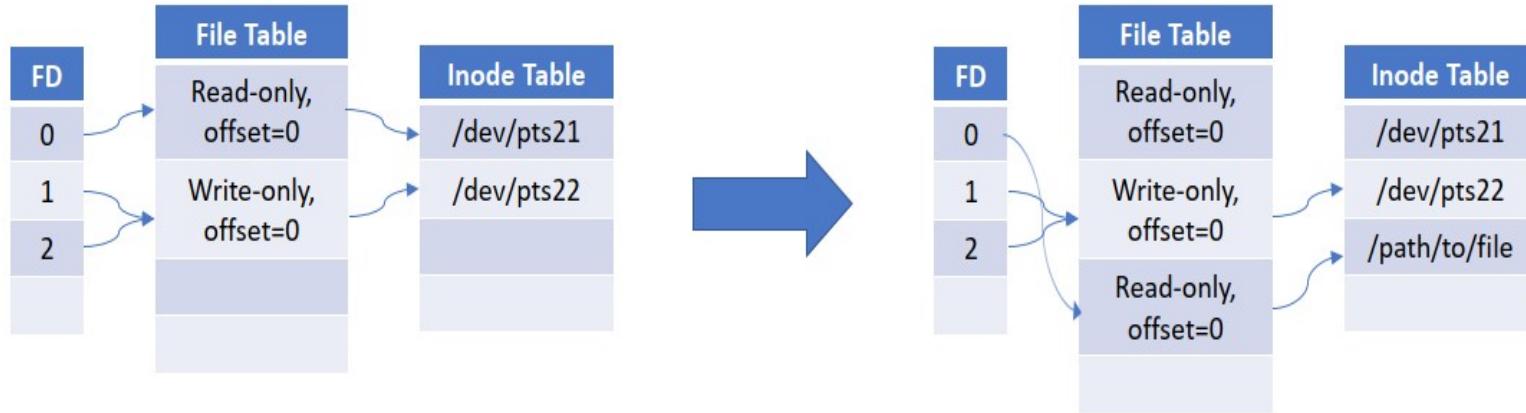
Example: Input Redirection

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



Example: Input Redirection

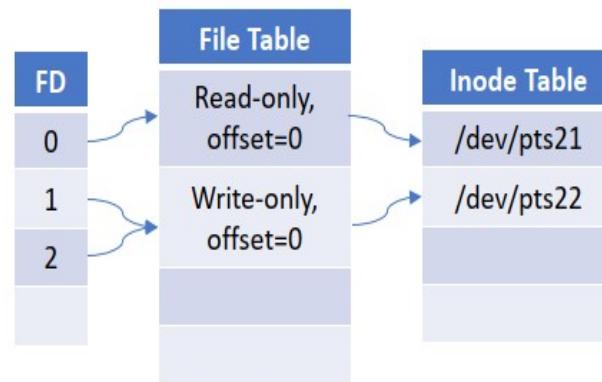
- Goal: instead of reading from stdin, read from another source



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

Example: Input Redirection

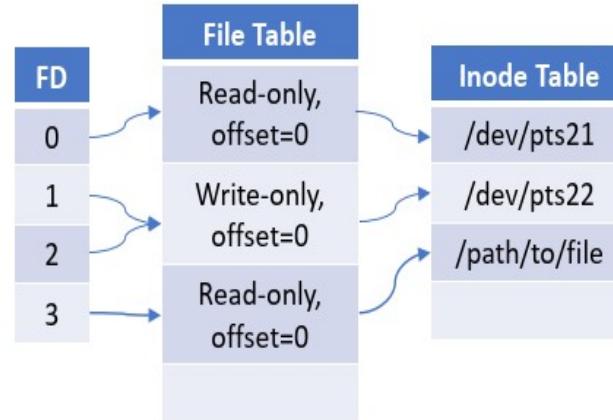
- Goal: instead of reading from stdin, read from another source



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

Example: Input Redirection

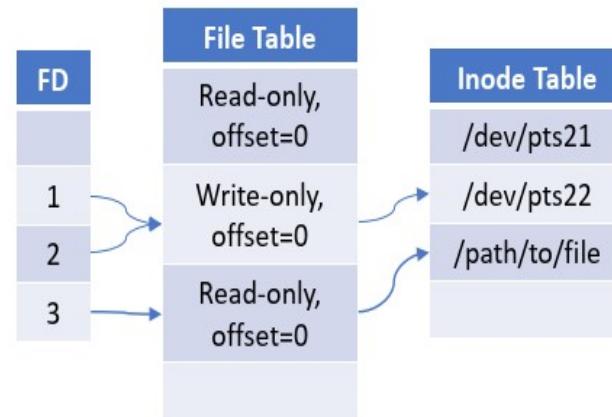
- Goal: instead of reading from stdin, read from another source



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

Example: Input Redirection

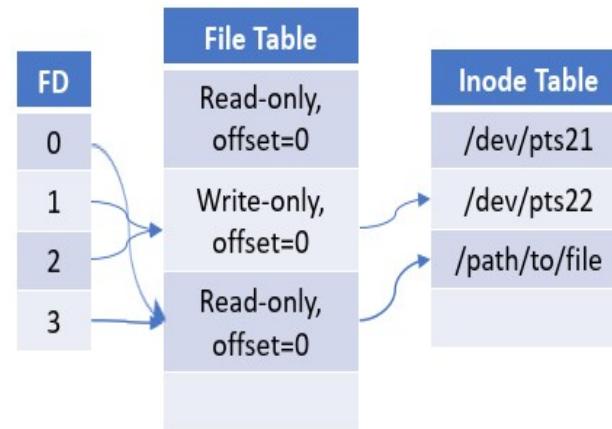
- Goal: instead of reading from stdin, read from another source



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

Example: Input Redirection

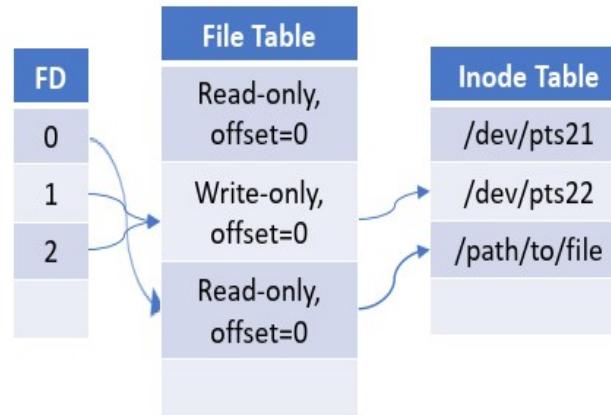
- Goal: instead of reading from stdin, read from another source



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

Example: Input Redirection

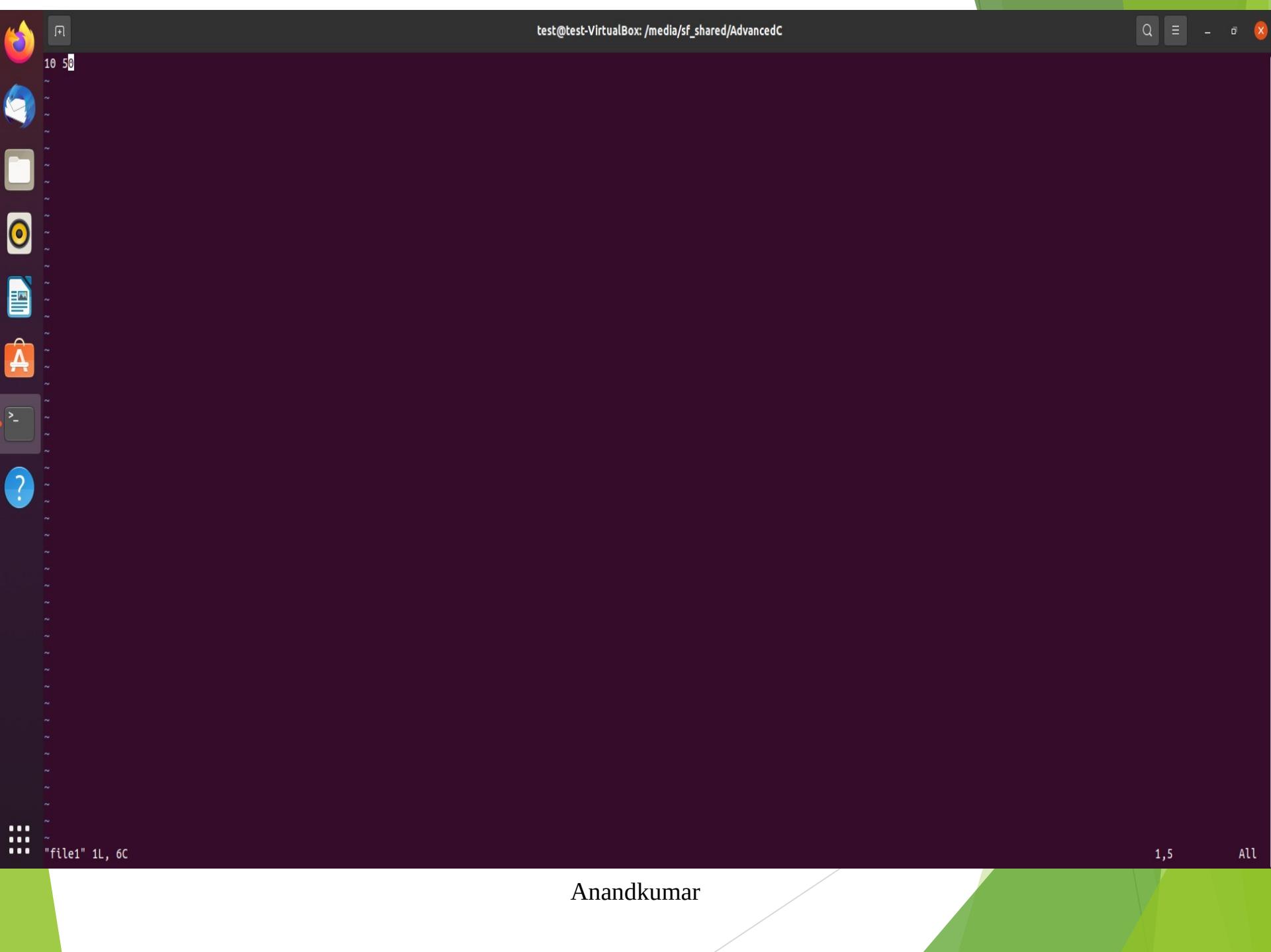
- Goal: instead of reading from stdin, read from another source



```
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);
}
```



Anandkumar



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

Q E - X

```
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$
```



Anandkumar

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.
- **GmailIFS**: Filesystem which stores data as mail in Gmail
- **EncFS**: Encrypted virtual filesystem

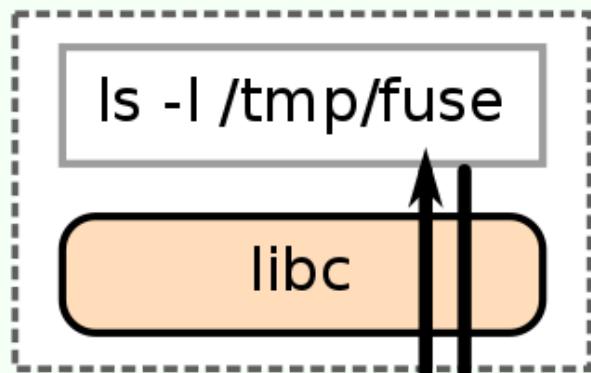
FUSE structure

- 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

Userspace



Kernel

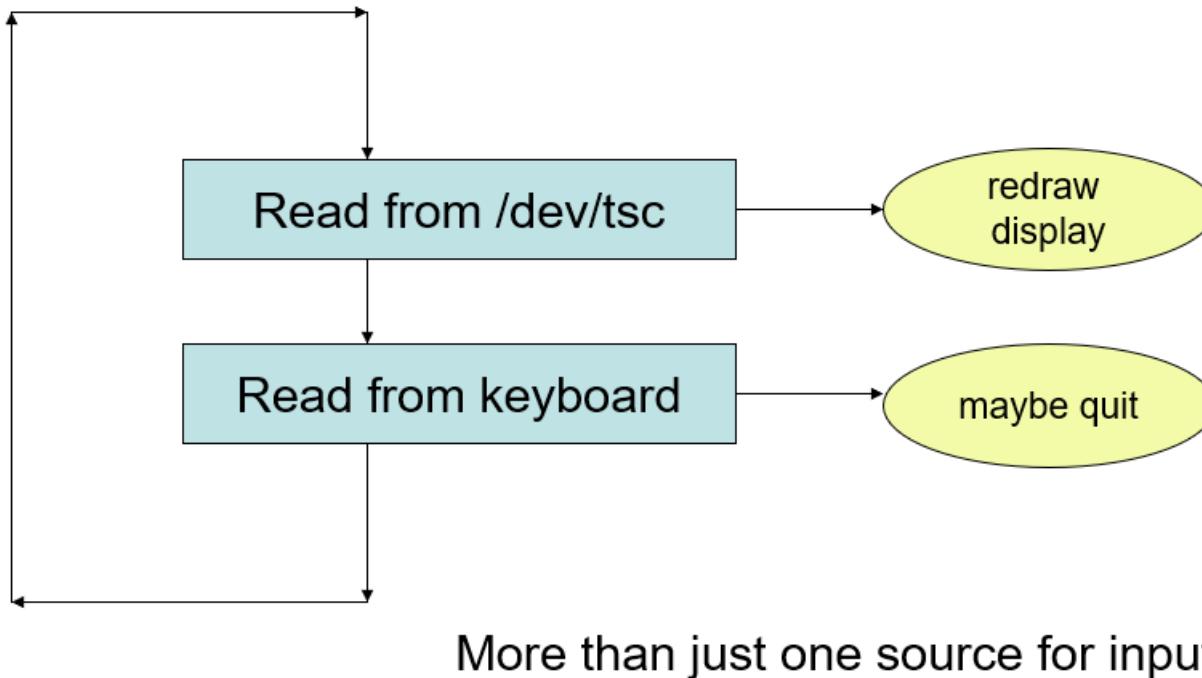


getattr	Get file attributes - similar to <code>stat()</code> .	chown	Change the owner and group of a file.
readlink	Read the target of a symbolic link.	truncate	Change the size of a file.
mknod	Create a file node.	open	Open a file.
mkdir	Create a directory.	release	Close an open file.
unlink	Remove a file.	read	Read data from an open file.
rmdir	Remove a directory.	write	Write data to an open file.
symlink	Create a symbolic link.	fsync	Synchronize file contents (ie. flush dirty buffers).
rename	Rename a file.	opendir	Open directory.
link	Create a hard link to a file.	readdir	Read directory - get directory listing.
chmod	Change the permission bits of a file.	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



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;
}
```

Anandkumar

]+

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 set for stderr
```

```
test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$ test@test-VirtualBox:/media/sf_shared/git/AdvancedC/FileNonBlock$ █
```

Anandkumar

```
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.

```
#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$ 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 */
};
```

```
#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$ gcc -g -o select1 select1.c
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$ 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.

POLLMMSG

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.

```
#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$ 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.

```
#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.

```
#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);
```

input.dat

20 30



x=20
y=30

Unlike `fseek()`, `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);
```

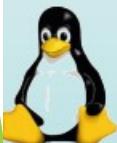
except that it also clears the error indicator.

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

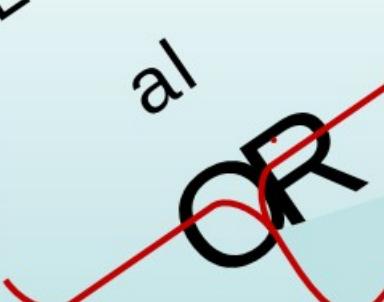
Facilities

- [LOG_AUTH](#)
- [LOG_AUTHPRIV](#)
- [LOG_CRON](#)
- [LOG_DAEMON](#)
- [LOG_FTP](#)
- [LOG_KERN](#)
- [LOG_LOCAL0-7](#)
- [LOG_LPR](#)
- [LOG_MAIL](#)
- [LOG_NEWS](#)
- [LOG_SYSLOG](#)
- [LOG_USER](#)
- [LOG_UUCP](#)

Levels

- [LOG_EMERG](#)
- [LOG_ALERT](#)
- [LOG_CRIT](#)
- [LOG_ERR](#)
- [LOG_WARNING](#)
- [LOG_NOTICE](#)
- [LOG_INFO](#)
- [LOG_DEBUG](#)

Logi
cal



Priority



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.



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

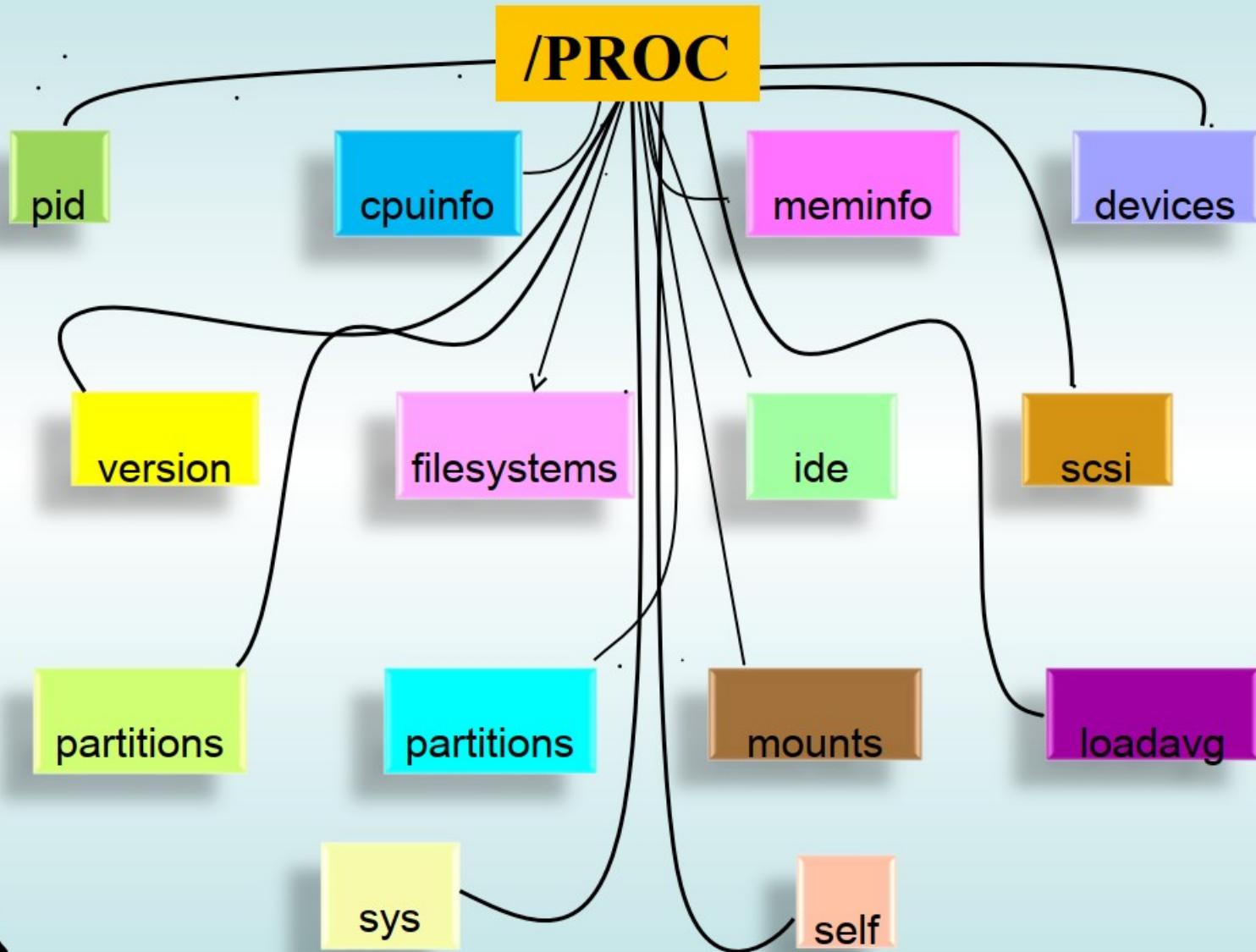
int main()
{
    setlogmask (LOG_UPTO (LOG_NOTICE));

    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$ 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

- 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 by 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: ~/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: ~/kernelmodule

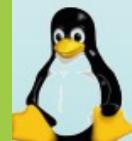
test@test-VirtualBox: /media/sf_shared/git/Advanc...

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
hypervisor    : 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 srbs
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

Program_1

Here goes the program_1 code

.

system(program_2)

.

.

.

Prgoram_2

Program_2 will run as
a command in Bourne
Shell

The remaining code will
continue executing.



test@test-VirtualBox: /media/sf_shar...

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;
}
```



Anandkumar



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

test@test-VirtualBox: ~/kernelmodule

test@test-VirtualBox: /media/sf_shared/git/Advanc...

test@test-VirtualBox: ~/kerne...

```
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

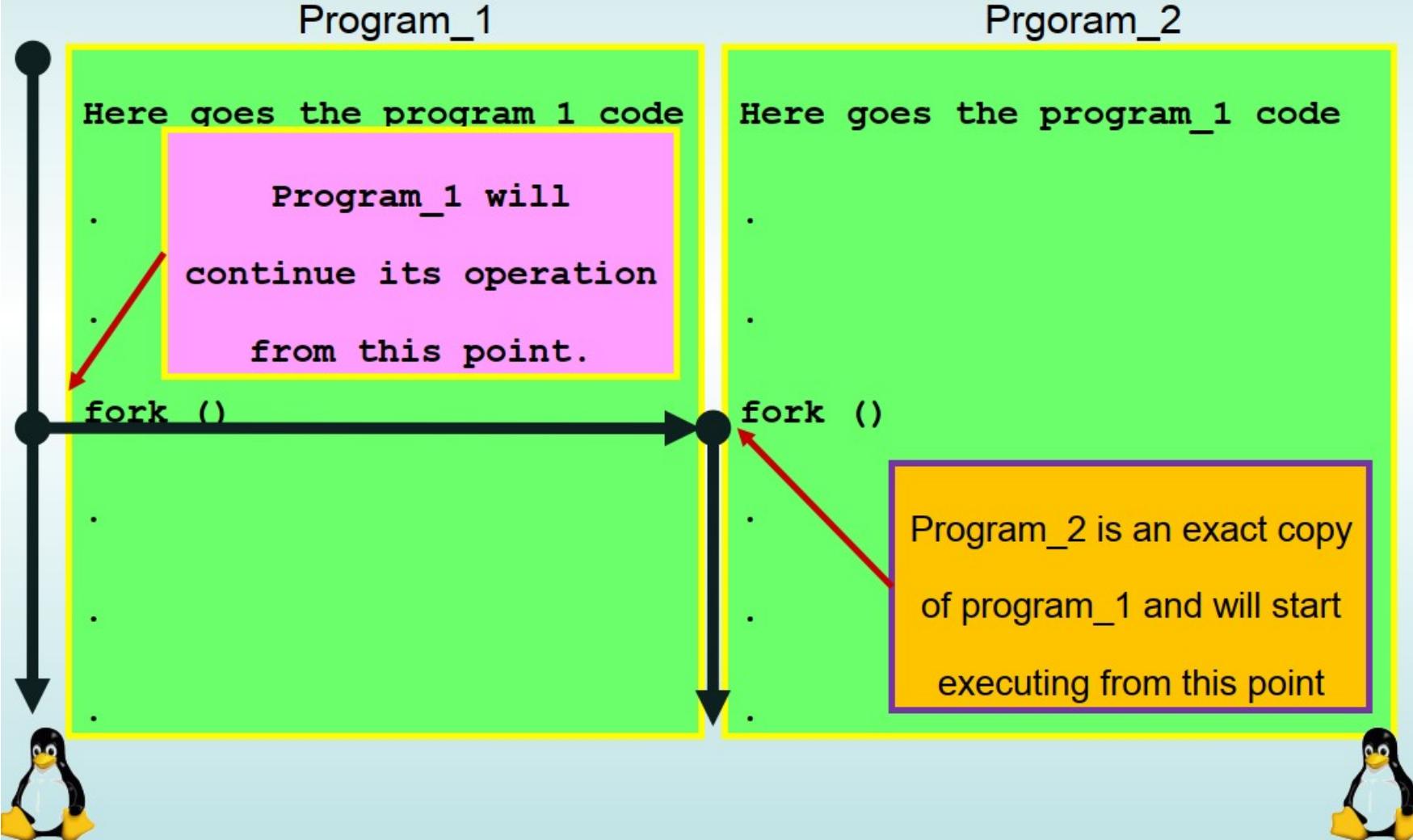
fork() returns

-1 if fork fails

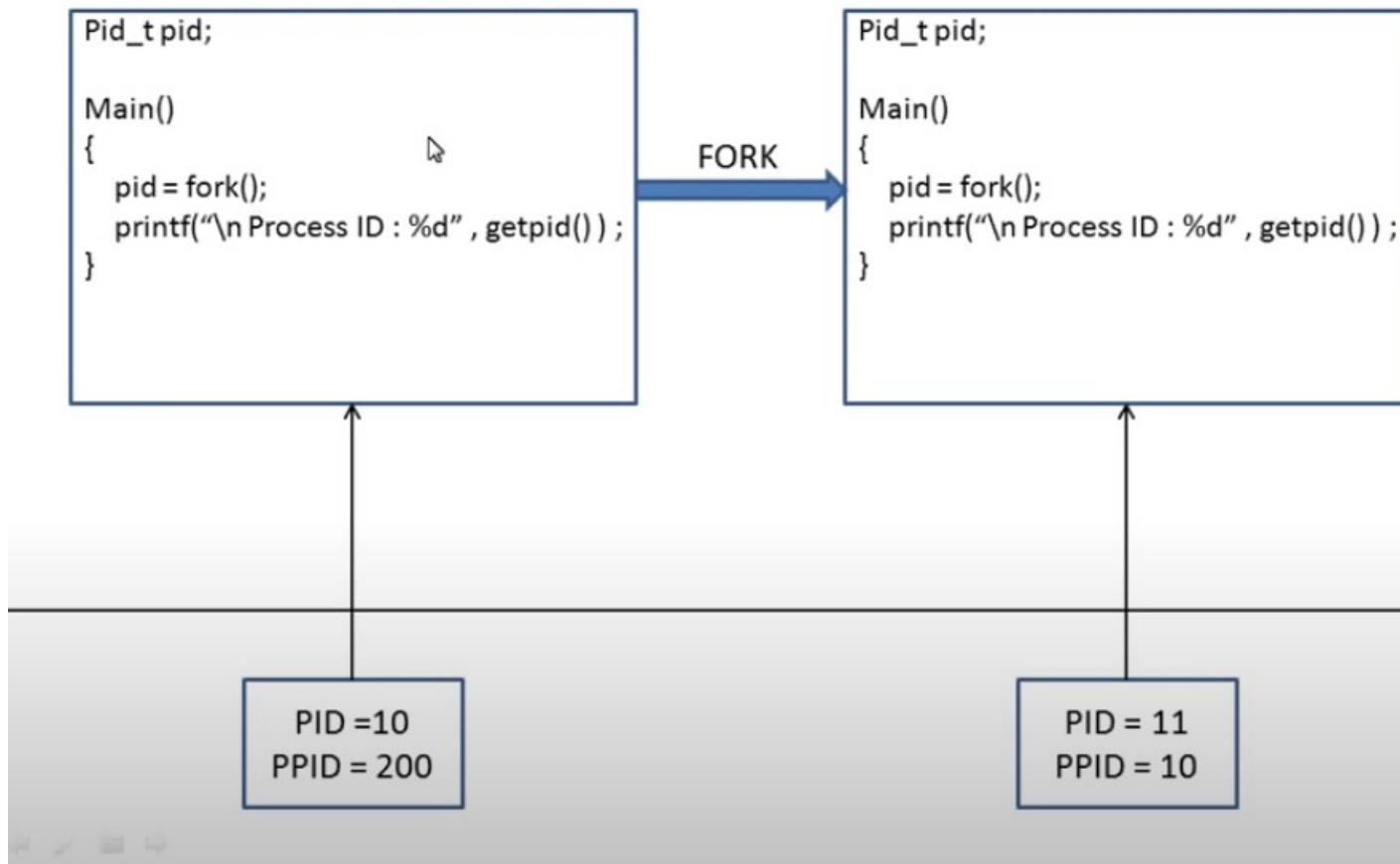
0 in child process

child's PID in parent process

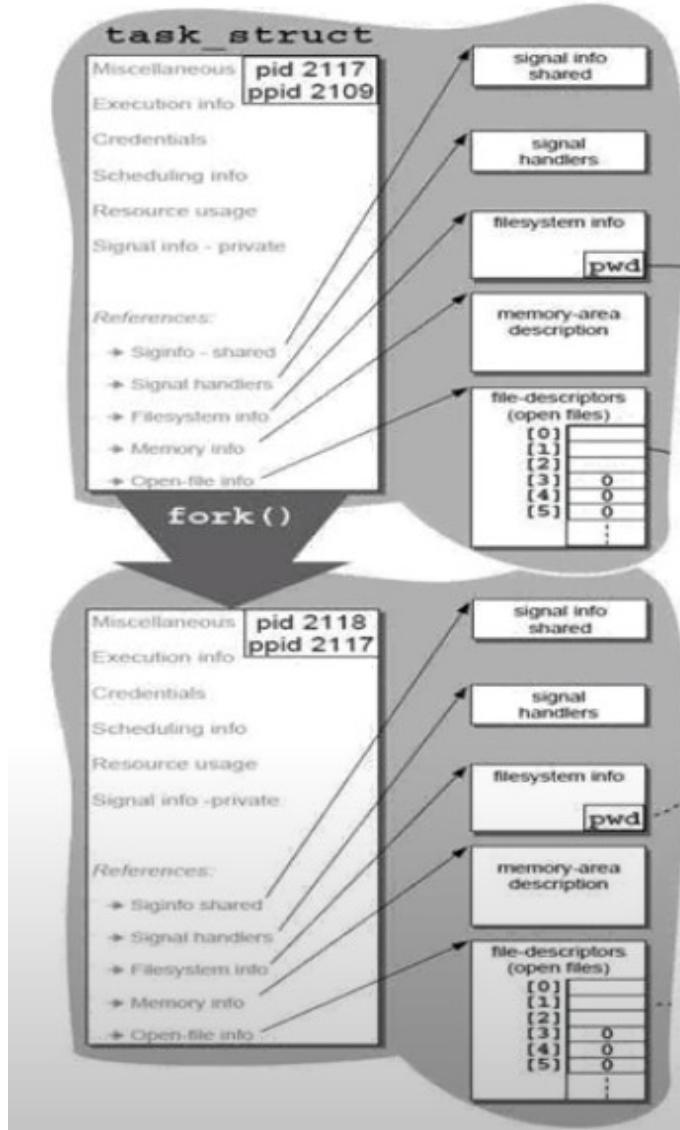
Creating Processes in Linux



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
 $nr_threads \geq max_threads$
- ✓ 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;
}
```

Anandkumar

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;
}
```

Anandkumar

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$ ./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;
}
```

```
#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$ 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$ ./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:/media/sf_shared/git/AdvancedC/ForkPid$
```

```
#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$ gcc -g -o pid_example1 pid_example1.c
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

Program_1

Here goes the program_1 code

exec (program_2)

Prgoram_2

Now, the program_2
will be replaced with
progaram_1 and run
till end.

The remaining program
code, will never execute if
'exec' finishes successfully.



Creating Processes in Linux

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



```
#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

<i>errno</i>	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.

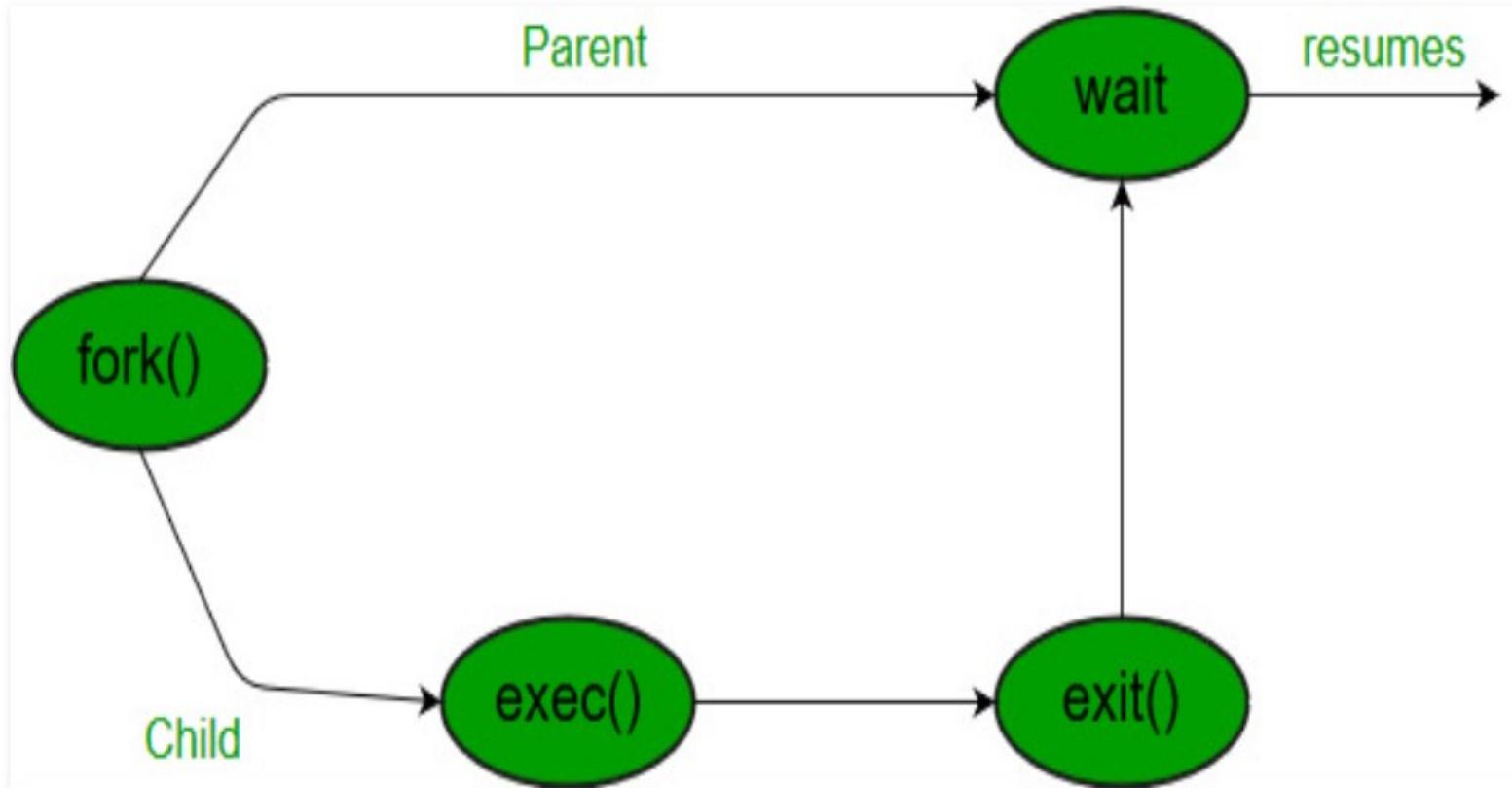
Waiting for a child to finish

(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);
```



```
#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;
}
```

```
#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;
}
```

```
#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;
}
```



```
#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

```
#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);
```

```
#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");
}
```

```
#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

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(void)
{
    printf("Before fork\n");
    vfork();
    printf("after fork\n");
}
```

```
#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`.



```
#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);                      /* stderr */

    /* do its daemon thing... */

    return 0;
}
```

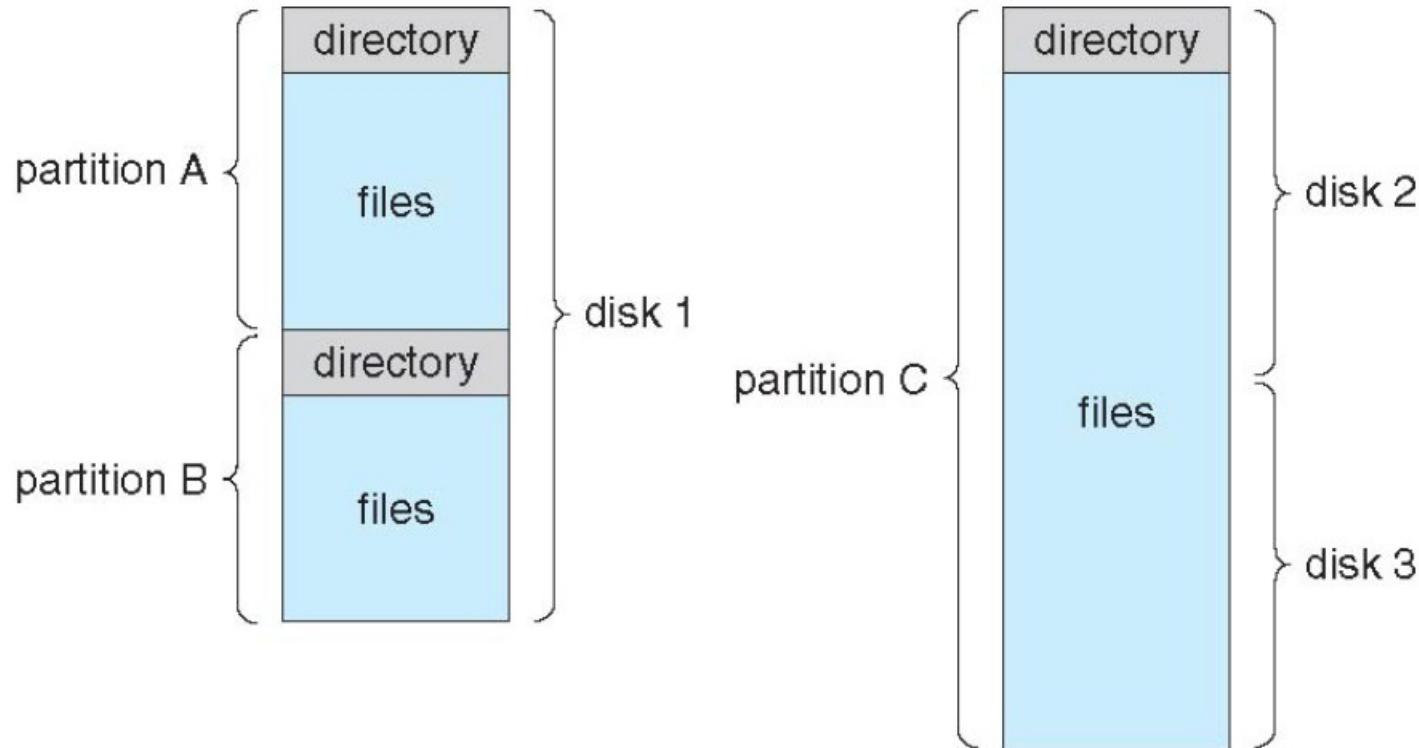
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

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

file permissions

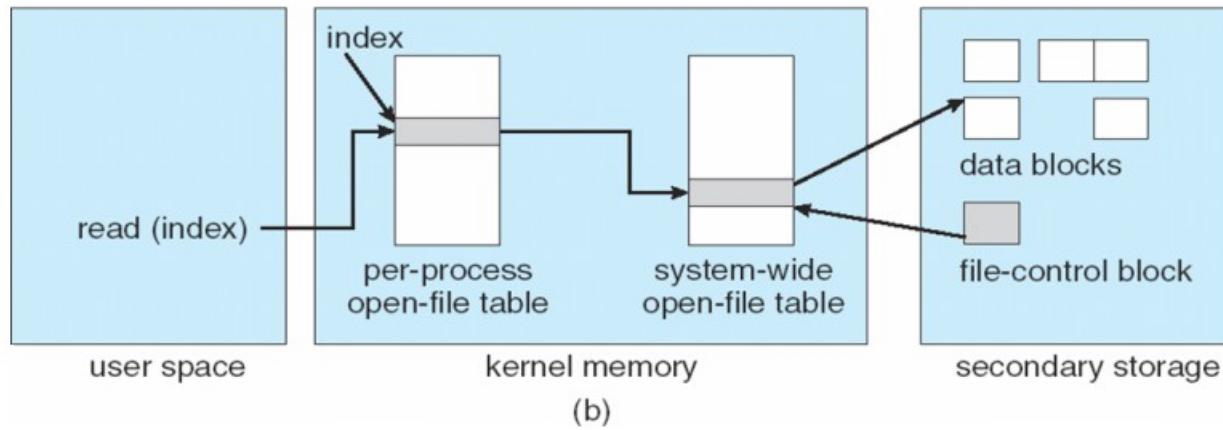
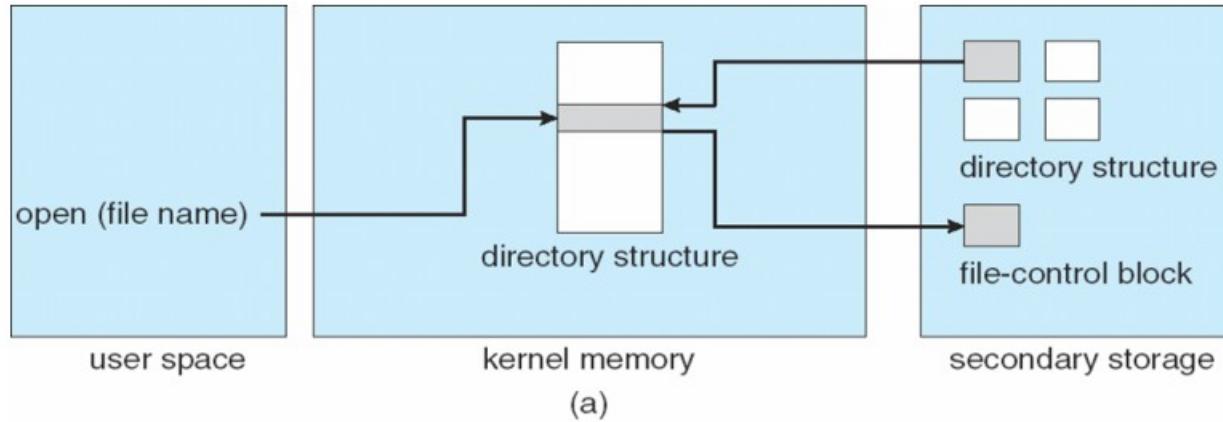
file dates (create, access, write)

file owner, group, ACL

file size

file data blocks or pointers to file data blocks

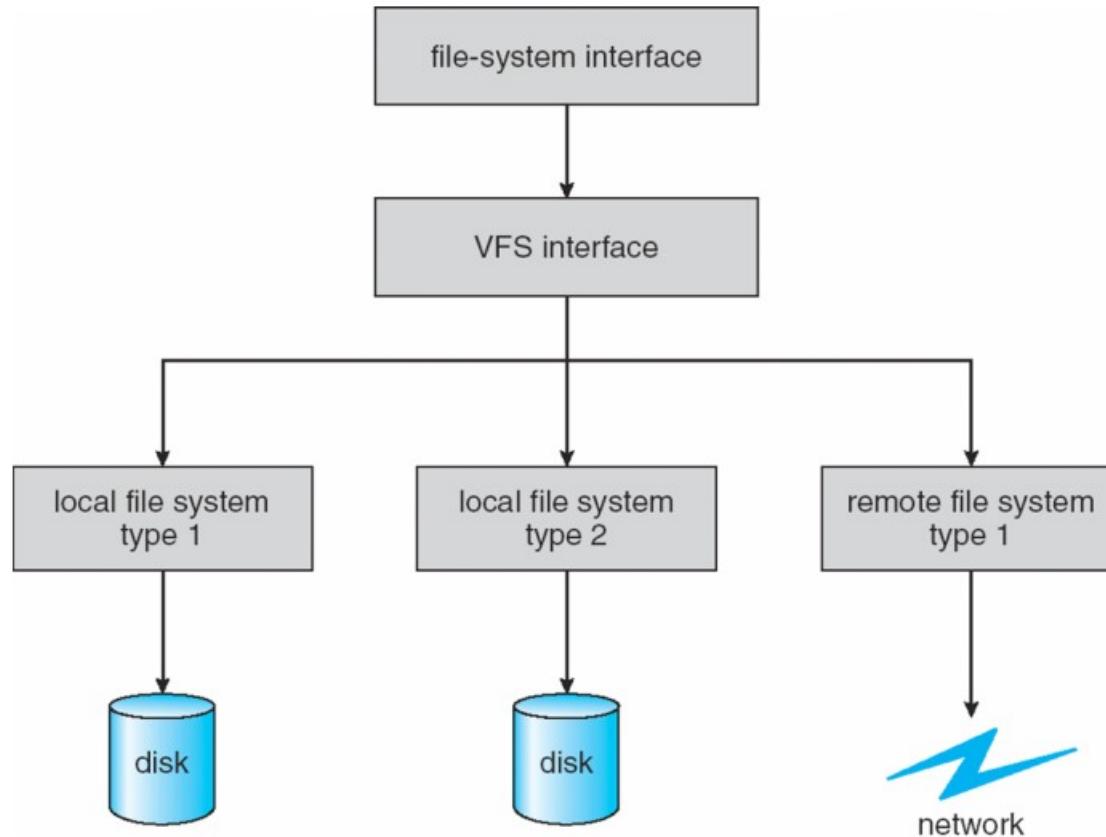
In-Memory File System Structures



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



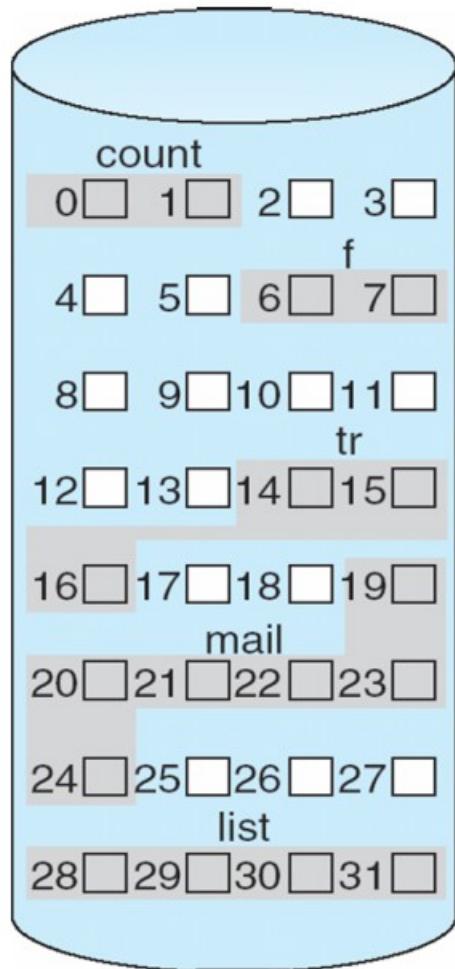
Allocation Methods

- 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

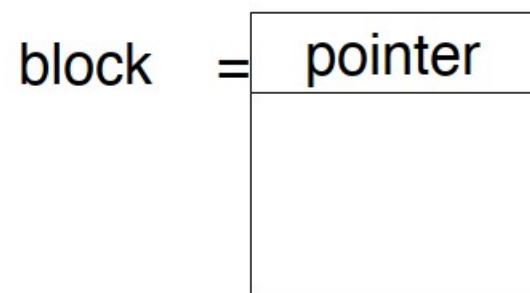


directory

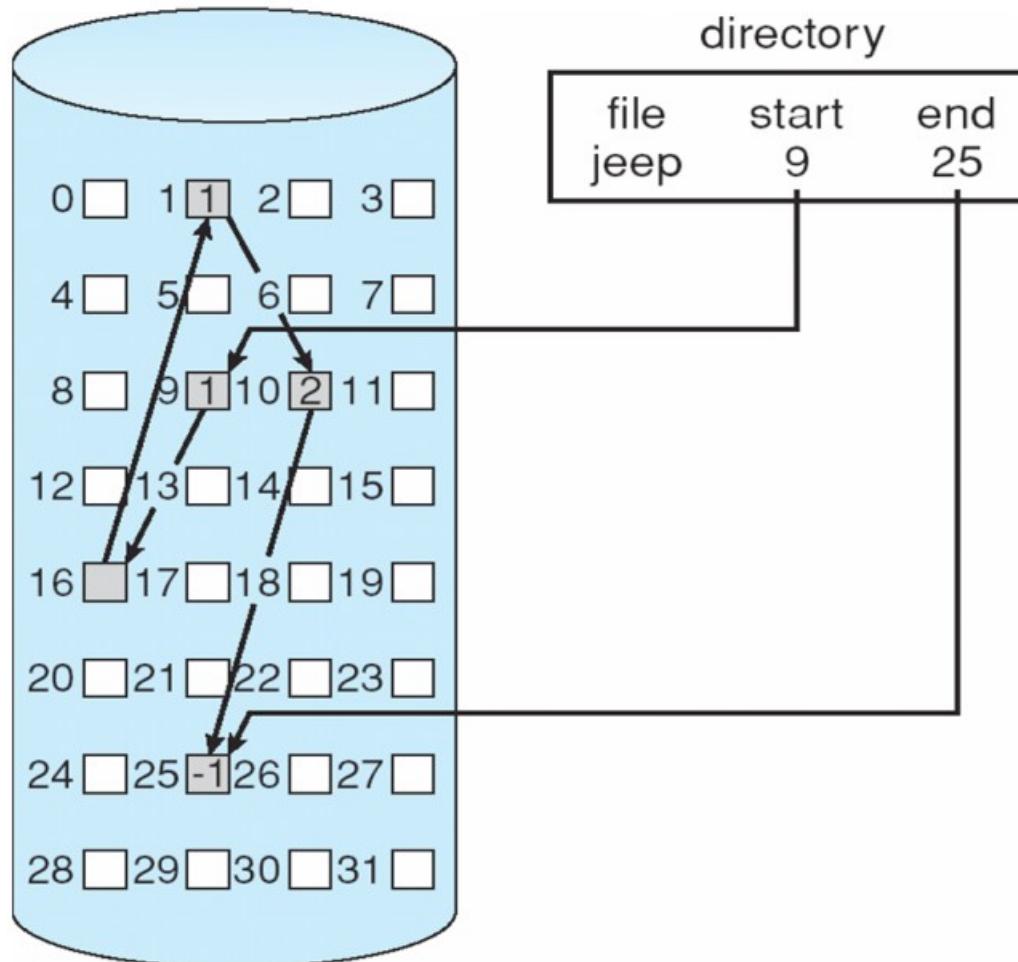
file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Linked Allocation

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

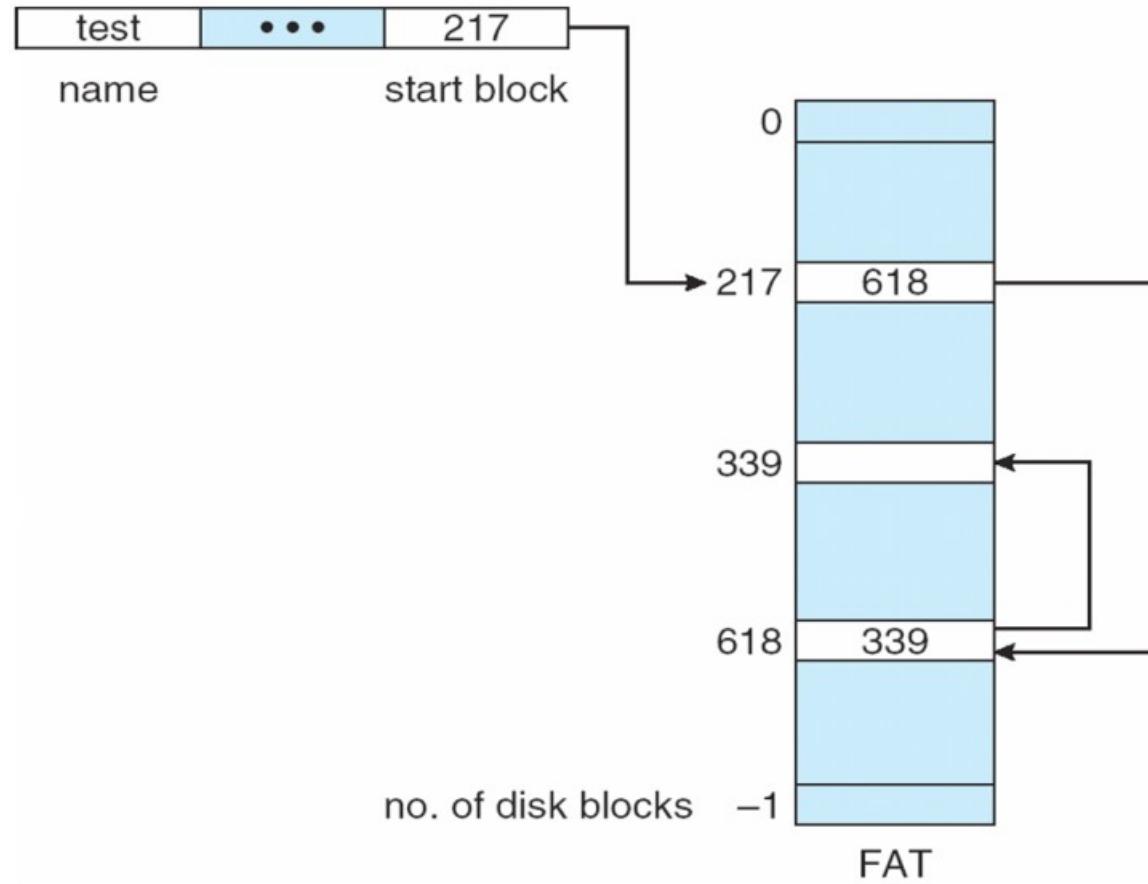


Linked Allocation



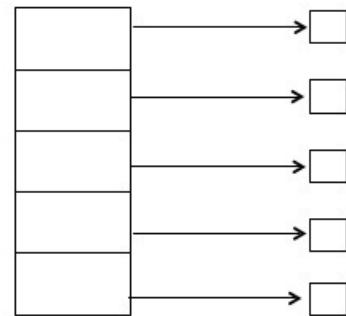
File-Allocation Table

directory entry



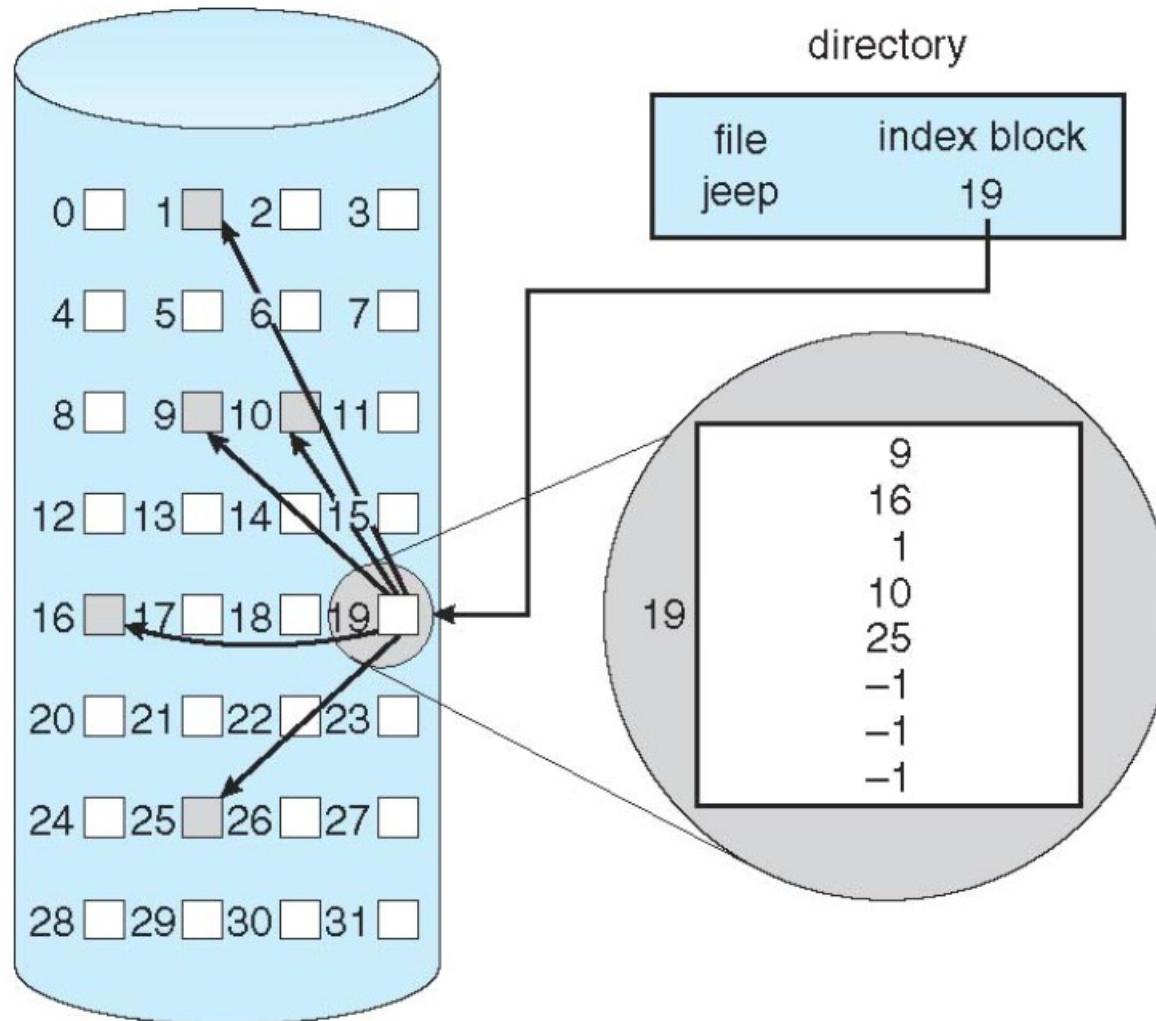
Indexed Allocation

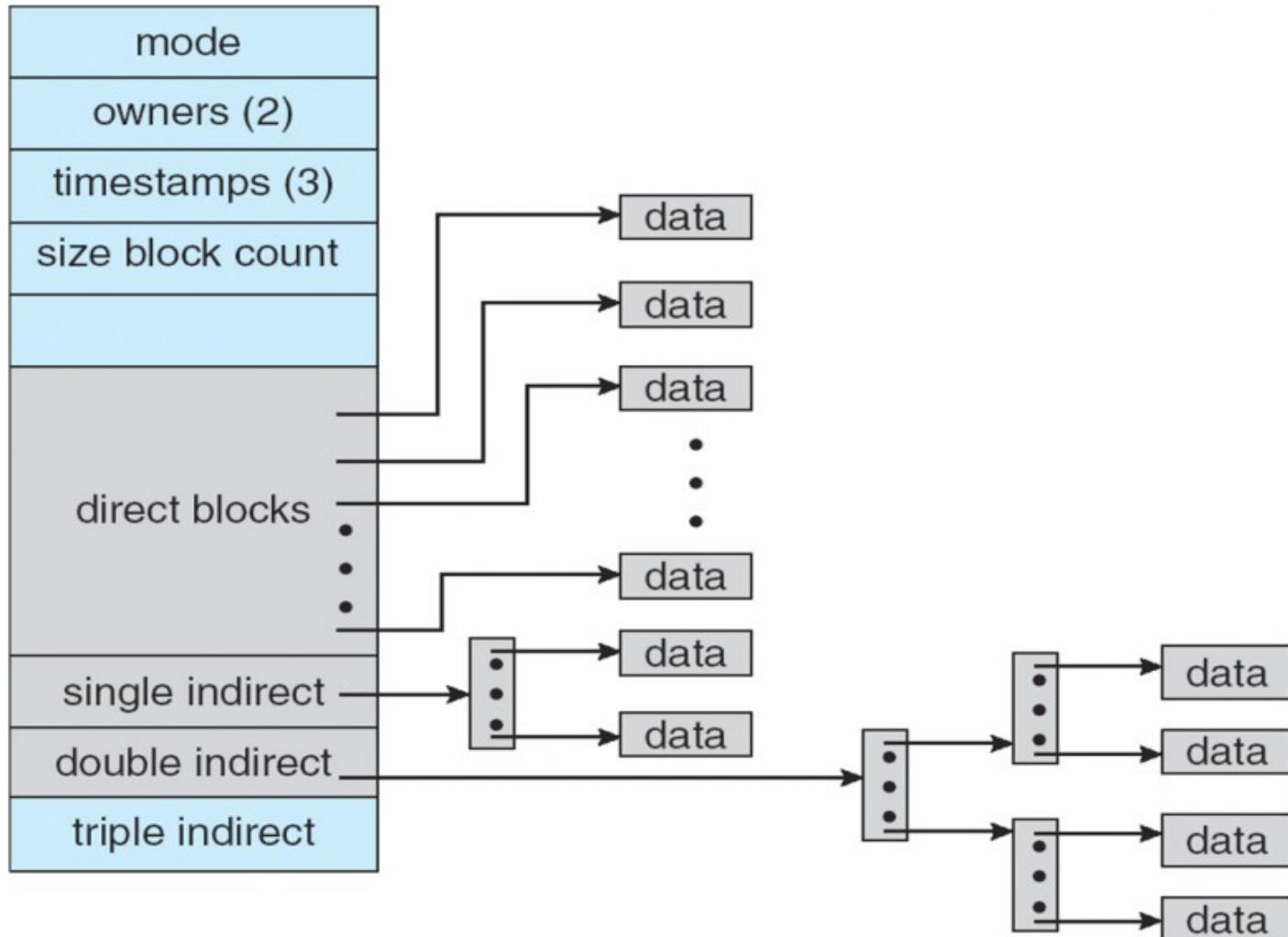
- Brings all pointers together into the **index block**
- Logical view



index table

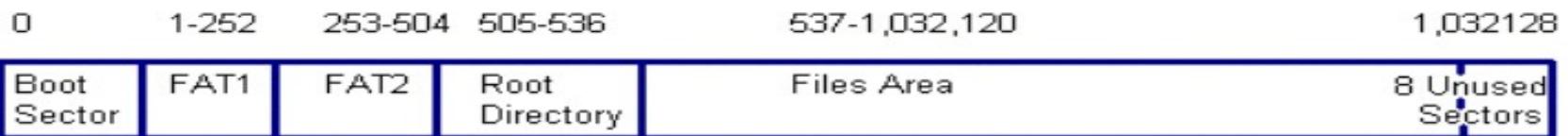
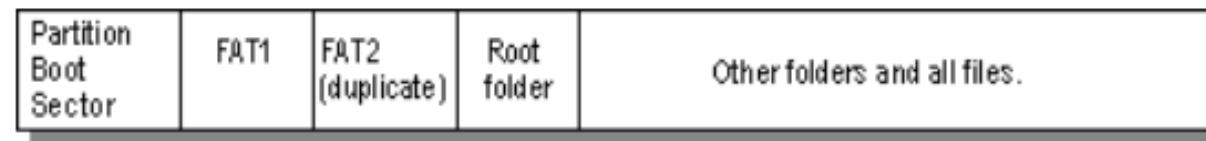
Example of Indexed Allocation





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
0xb	25B		BIOS Parameter Block
0x24	26B		Extended Bios Parameter Block
0xe	448 B		Bootstrap Code
0xfe	2B	55 aa	End of Sector Marker

Offset	Length of Field	Typical Value(in hex)	Meaning
0xb, 11	2B	00 02	The number of bytes, in big-endian. The typical value of 0002 translates to 0x0200 = 512
0xd, 13	1B	01	Number of Sectors per Cluster.
0xe, 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

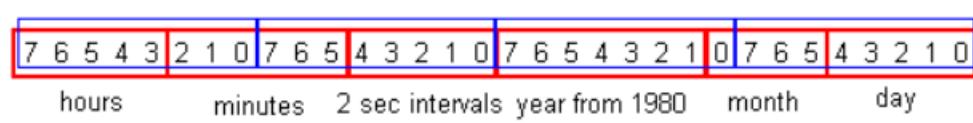
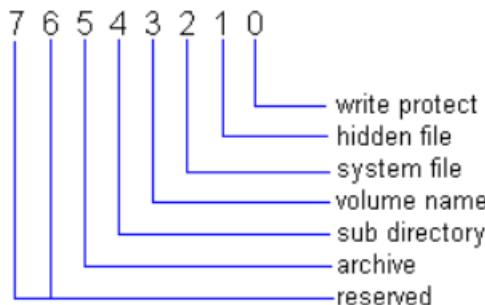
FAT 12	FAT 16	Meaning
000	0000	available for allocation
001	0001	not used
FF0	FFF0-FFF6	reserved
FF7	FFF7	bad cluster, dna
FF8-FFF	FFF8-FFFF	last cluster in file
everything else	everything else	next cluster

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.



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*