Learning Kernel Programming [closed]

I want to learn linux kernel programming.

What would be the starting points for that ? What could be some of the simpler problems to target ?

thanks in advance

linux kernel linux-device-driver embedded-linux

edited Jan 5 at 14:24
manav m-n
2,968 6 34 70

asked May 27 '09 at 8:54
Geek
4,797 10 41 59

closed as off-topic by bummi, Willie Wheeler, Pang, Macmade, phresnel Jan 8 at 7:36

This question appears to be off-topic. The users who voted to close gave this specific reason:

- "Questions asking us to **recommend or find a book, tool, software library, tutorial or other off-site resource** are off-topic for Stack Overflow as they tend to attract opinionated answers and spam. Instead, describe the problem and what has been done so far to solve it." – bummi, Willie Wheeler, Macmade

If this question can be reworded to fit the rules in the help center, please edit the question.

7 Answers

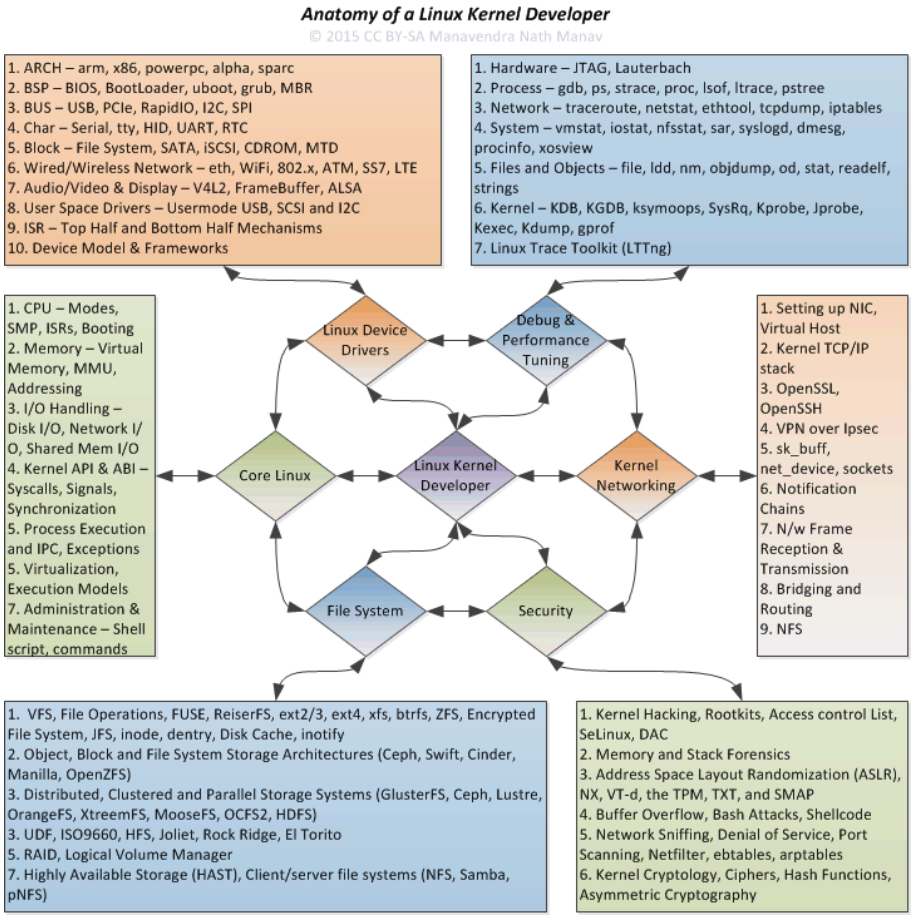
Try to get hold of Robert Love's book on Linux Kernel Programming. Its very concise and easy to follow.

After that or along with that, you may want to take a look at "Understanding the Linux kernel".But I wouldn't recommend it during the early stages.

Also, look at the Linux kernel programming guide. Since a lot can be learnt from programing kernel modules, that guide will help you. And yes, for a lot of information, consult the 'documentation' sub-directory of the Kernel sources tarball.

edited Feb 3 '14 at 19:41
animuson ♦
27.3k 18 69 109

answered May 27 '09 at 9:05
Amit
6,324 8 34 56




```
**TODO** +editPic: Linux Kernel Developer -> (Ring Layer 0)
         +addSection: Kernel Virtualization Engine

KERN_WARN_CODING_STYLE: Do not Loop unless you absolutely have to.
```

> **Recommended Books for the *Uninitialized* `void *i`**
>
> *"Men do not understand books until they have a certain amount of life, or at any rate no man understands a deep book, until he has seen and lived at least part of its contents". –Ezra Pound*

A journey of a thousand *code-miles* must begin with a single step. If you are in confusion about which of the following books to start with, don't worry, pick any one of your choice. Not all those who wander are lost. As *all roads ultimately connect to highway*, you will explore new things in your kernel journey as the pages progress without meeting any dead ends, and ultimately connect to the `code-set` . Read with alert mind and remember: ***Code is not Literature***.

What is left is not a thing or an emotion or an image or a mental picture or a memory or even an idea. It is a function. A process of some sort. An aspect of Life that could be described as a function of something "larger". And therefore, it appears that it is not really "separate" from that something else. Like the function of a knife - cutting something - is not, in fact, separate from the knife itself. The function may or may not be in use at the moment, but it is potentially NEVER separate.

Solovay Strassen Derandomized Algorithm for Primality Test:

```
input odd integer n ≥ 3
for a = 2 to 2(log n)^2:
      if gcd(a, n) ≠ 1:
          return 'composite'
      else if (a/n) ≢ a^((n-1)/2) (mod n):
          return 'composite'
return 'prime'
```

Read not to contradict and confute; nor to believe and take for granted; nor to find talk and discourse; but to weigh and consider. Some books are to be tasted, others to be swallowed, and some few to be chewed and digested: that is, some books are to be read only in parts, others to be read, but not curiously, and some few to be read wholly, and with diligence and attention.

```
static void tasklet_hi_action(struct softirq_action *a)
{
        struct tasklet_struct *list;

        local_irq_disable();
        list = __this_cpu_read(tasklet_hi_vec.head);
        __this_cpu_write(tasklet_hi_vec.head, NULL);
        __this_cpu_write(tasklet_hi_vec.tail, this_cpu_ptr(&tasklet_hi_vec.head));
        local_irq_enable();

        while (list) {
                struct tasklet_struct *t = list;

                list = list->next;

                if (tasklet_trylock(t)) {
                        if (!atomic_read(&t->count)) {
                                if (!test_and_clear_bit(TASKLET_STATE_SCHED,
                                                        &t->state))
                                        BUG();
                                t->func(t->data);
                                tasklet_unlock(t);
                                continue;
                        }
                        tasklet_unlock(t);
                }

                local_irq_disable();
                t->next = NULL;
                *__this_cpu_read(tasklet_hi_vec.tail) = t;
                __this_cpu_write(tasklet_hi_vec.tail, &(t->next));
                __raise_softirq_irqoff(HI_SOFTIRQ);
                local_irq_enable();
        }
}
```

> **Core Linux** (5 -> 1 -> 3 -> 2 -> 7 -> 4 -> 6)
>
> *“Nature has neither kernel nor shell; she is everything at once” -- Johann Wolfgang von Goethe*

Reader should be well versed with operating system concepts; a fair understanding of long running processes and its differences with processes with short bursts of execution; fault tolerance while meeting soft and hard real time constraints. While reading, it's important to understand and `n/ack` the design choices made by the linux kernel source in the core subsystems.

Threads [and] signals [are] a platform-dependent trail of misery, despair, horror and madness (~Anthony Baxte). That being said you should be a self-evaluating C expert, before diving into the kernel. You should also have good experience with Linked Lists, Stacks, Queues, Red Blacks Trees, Hash Functions, et al.

```
volatile int i;
int main(void)
{
    int c;
    for (i=0; i<3; i++) {
        c = i&&i;
        printf("%d\n", c);    /* find c */
    }
    return 0;
}
```

The beauty and art of the Linux Kernel source lies in the deliberate code obfuscation used along. This is often necessitated as to convey the computational meaning involving two or more operations in a clean and elegant way. This is especially true when writing code for multi-core architecture.

Video Lectures on Real-Time Systems, Task Scheduling, Memory Compression, Memory Barriers, SMP

```
#ifdef __compiler_offsetof
#define offsetof(TYPE,MEMBER) __compiler_offsetof(TYPE,MEMBER)
#else
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
#endif
```

1. Linux Kernel Development - *Robert Love*
2. Understanding the Linux Kernel - *Daniel P. Bovet, Marco Cesati*
3. The Art of Linux KerneL Design - *Yang Lixiang*

4. Professional Linux Kernel Architecture - *Wolfgang Mauerer*
5. Design of the UNIX Operating System - *Maurice J. Bach*
6. Understanding the Linux Virtual Memory Manager - *Mel Gorman*
7. Linux Kernel Internals - *Tigran Aivazian*
8. Embedded Linux Primer - *Christopher Hallinan*

> **Linux Device Drivers** (1 -> 2 -> 4 -> 3 -> 8 -> ...)
>
> *"Music does not carry you along. You have to carry it along strictly by your ability to really just focus on that little small kernel of emotion or story". -- Debbie Harry*

Your task is basically to establish a high speed communication interface between the hardware device and the software kernel. You should read the hardware reference datasheet/manual to understand the behavior of the device and it's control and data states and provided physical channels. Knowledge of Assembly for your particular architecture and a fair knowledge of VLSI Hardware Description Languages like VHDL or Verilog will help you in the long run.

- Intel® 64 and IA-32 Architectures Software Developer's Manual
- ARM Architecture Reference Manual
- ARM System Developer's Guide

Q: But, why do I have to read the hardware specs?

A: Because, "There is a chasm of carbon and silicon the software can't bridge" - Rahul Sonnad

However, the above doesn't poses a problem for *Computational Algorithms* (Driver code - *bottom-half processing*), as it can be fully simulated on a Universal Turing Machine. If the computed result holds true in the mathematical domain, it's a certainty that it is also true in the physical domain.

Video Lectures on Linux Device Drivers (Lec. 17 & 18), Anatomy of an Embedded KMS Driver, Pin Control and GPIO Update, Common Clock Framework, Write a Real Linux Driver - Greg KH

```
static irqreturn_t phy_interrupt(int irq, void *phy_dat)
{
         struct phy_device *phydev = phy_dat;

         if (PHY_HALTED == phydev->state)
                 return IRQ_NONE;               /* It can't be ours.  */

         /* The MDIO bus is not allowed to be written in interrupt
          * context, so we need to disable the irq here.  A work
          * queue will write the PHY to disable and clear the
          * interrupt, and then reenable the irq line.
          */
         disable_irq_nosync(irq);
         atomic_inc(&phydev->irq_disable);

         queue_work(system_power_efficient_wq, &phydev->phy_queue);

         return IRQ_HANDLED;
}
```

1. Linux Device Drivers - *Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman*
2. Essential Linux Device Drivers - *Sreekrishnan Venkateswaran*
3. Writing Linux Device Drivers - *Jerry Cooperstein*
4. The Linux Kernel Module Programming Guide - *Peter Jay Salzman, Michael Burian, Ori Pomerantz*
5. Linux PCMCIA Programmer's Guide - *David Hinds*
6. Linux SCSI Programming Howto - *Heiko Eibfeldt*
7. Serial Programming Guide for POSIX Operating Systems - *Michael R. Sweet*
8. Linux Graphics Drivers: an Introduction - *Stéphane Marchesin*
9. Programming Guide for Linux USB Device Drivers - *Detlef Fliegl*
10. The Linux Kernel Device Model - *Patrick Mochel*

> **Kernel Networking** (1 -> 2 -> 3 -> ...)
>
> *"Call it a clan, call it a network, call it a tribe, call it a family: Whatever you call it, whoever you*

> *are, you need one." - Jane Howard*

Understanding a packet walk-through in the kernel is a key to understanding kernel networking. Understanding it is a must if we want to understand Netfilter or IPSec internals, and more. The two most important structures of linux kernel network layer are: `struct sk_buff` and `struct net_device`

```
static inline int sk_hashed(const struct sock *sk)
{
        return !sk_unhashed(sk);
}
```

1. Understanding Linux Network Internals - *Christian Benvenuti*
2. Linux Kernel Networking: Implementation and Theory - *Rami Rosen*
3. UNIX Network Programming - *W. Richard Stevens*
4. The Definitive Guide to Linux Network Programming - *Keir Davis, John W. Turner, Nathan Yocom*
5. The Linux TCP/IP Stack: Networking for Embedded Systems - *Thomas F. Herbert*
6. Linux Socket Programming by Example - *Warren W. Gay*
7. Linux Advanced Routing & Traffic Control HOWTO - *Bert Hubert*

> **Kernel Debugging** (1 -> 4 -> 9 -> ...)
>
> *Unless in communicating with it one says exactly what one means, trouble is bound to result. ~Alan Turing, about computers*

Brian W. Kernighan, in the paper Unix for Beginners (1979) said, "The most effective debugging tool is still careful thought, coupled with judiciously placed print statements". Knowing what to collect will help you to get the right data quickly for a fast diagnosis. The great computer scientist Edsger Dijkstra once said that testing can demonstrate the presence of bugs but not their absence. Good investigation practices should balance the need to solve problems quickly, the need to build your skills, and the effective use of subject matter experts.

Video Lectures on Kernel Debug and Profiling, Core Dump Analysis, Multicore Debugging with GDB, Controlling Multi-Core Race Conditions, Debugging Electronics

```
/* Buggy Code -- Stack frame problem
 * If you require information, do not free memory containing the information
 */
char *initialize() {
  char string[80];
  char* ptr = string;
  return ptr;
}

int main() {
  char *myval = initialize();
  do_something_with(myval);
}
/*  "When debugging, novices insert corrective code; experts remove defective code."
 *     - Richard Pattis
#if DEBUG
 printk("The above can be considered as Development and Review in Industrial Practises");
#endif
 */
```

1. Linux Debugging and Performance Tuning - *Steve Best*
2. Linux Applications Debugging Techniques - *Aurelian Melinte*
3. Debugging with GDB: The GNU Source-Level Debugger - *Roland H. Pesch*
4. Debugging Embedded Linux - *Christopher Hallinan*
5. The Art of Debugging with GDB, DDD, and Eclipse - *Norman S. Matloff*
6. Why Programs Fail: A Guide to Systematic Debugging - *Andreas Zeller*
7. Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code - *Bill Blunden*
8. Debugging: Finding most Elusive Software and Hardware Problems - *David J. Agans*
9. Debugging by Thinking: A Multidisciplinary Approach - *Robert Charles Metzger*
10. Find the Bug: A Book of Incorrect Programs - *Adam Barr*

> **File Systems** (1 -> 2 -> 6 -> ...)

> *"I wanted to have virtual memory, at least as it's coupled with file systems". -- Ken Thompson*

On a UNIX system, everything is a file; if something is not a file, it is a process, except for named pipes and sockets. In a file system, a file is represented by an `inode` , a kind of serial number containing information about the actual data that makes up the file. The Linux Virtual File System `VFS` caches information in memory from each file system as it is mounted and used. A lot of care must be taken to update the file system correctly as data within these caches is modified as files and directories are created, written to and deleted. The most important of these caches is the Buffer Cache, which is integrated into the way that the individual file systems access their underlying block storage devices.

Video Lectures on Storage Systems, Flash Friendly File System

```
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
        struct open_flags op;
        int fd = build_open_flags(flags, mode, &op);
        struct filename *tmp;

        if (fd)
                return fd;

        tmp = getname(filename);
        if (IS_ERR(tmp))
                return PTR_ERR(tmp);

        fd = get_unused_fd_flags(flags);
        if (fd >= 0) {
                struct file *f = do_filp_open(dfd, tmp, &op);
                if (IS_ERR(f)) {
                        put_unused_fd(fd);
                        fd = PTR_ERR(f);
                } else {
                        fsnotify_open(f);
                        fd_install(fd, f);
                }
        }
        putname(tmp);
        return fd;
}

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;

        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

1. Linux File Systems - *Moshe Bar*
2. Linux Filesystems - *William Von Hagen*
3. UNIX Filesystems: Evolution, Design, and Implementation - *Steve D. Pate*
4. Practical File System Design - *Dominic Giampaolo*
5. File System Forensic Analysis - *Brian Carrier*
6. Linux Filesystem Hierarchy - *Binh Nguyen*
7. BTRFS: The Linux B-tree Filesystem - *Ohad Rodeh*
8. StegFS: A Steganographic File System for Linux - *Andrew D. McDonald, Markus G. Kuhn*

> **Security** (1 -> 2 -> 8 -> 4 -> 3 -> ...)
>
> *"UNIX was not designed to stop its users from doing stupid things, as that would also stop them from doing clever things". — Doug Gwyn*
>
> No technique works if it isn't used. Ethics change with technology.
>
> "***F* × *S* = *k***" the product of freedom and security is a constant. - Niven's Laws

Cryptography forms the basis of trust online. Hacking is exploiting security controls either in a technical, physical or a human-based element. Protecting the kernel from other running programs is a first step toward a secure and stable system, but this is obviously not enough: some degree of protection must exist between different user-land applications as well. Exploits can target local or remote services.

> “You can't hack your destiny, brute force...you need a back door, a side channel into Life." — *Clyde Dsouza*

> Computers do not solve problems, they execute solutions. Behind every non-deterministic algorithmic code, there is a determined mind. -- */var/log/dmesg*

Video Lectures on Cryptography and Network Security, Namespaces for Security, Protection Against Remote Attacks, Secure Embedded Linux

```
env x='() { :;}; echo vulnerable' bash -c "echo this is a test for Shellsock"
```

1. Hacking: The Art of Exploitation - *Jon Erickson*
2. The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System - *Bill Blunden*
3. Hacking Exposed: Network Security Secrets - *Stuart McClure, Joel Scambray, George Kurtz*
4. A Guide to Kernel Exploitation: Attacking the Core - *Enrico Perla, Massimiliano Oldani*
5. The Art of Memory Forensics - *Michael Hale Ligh, Andrew Case, Jamie Levy, AAron Walters*
6. Practical Reverse Engineering - *Bruce Dang, Alexandre Gazet, Elias Bachaalany*
7. Practical Malware Analysis - *Michael Sikorski, Andrew Honig*
8. Maximum Linux Security: A Hacker's Guide to Protecting Your Linux Server - *Anonymous*
9. Linux Security - *Craig Hunt*
10. Real World Linux Security - *Bob Toxen*

> **Kernel Source** (0.11 -> 2.4 -> 2.6 -> 3.18)
>
> "Like wine, the mastery of kernel programming matures with time. But, unlike wine, it gets sweeter in the process". --Lawrence Mucheka

You might not think that programmers are artists, but programming is an extremely creative profession. It's logic-based creativity. Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter. As you already know, there is a difference between knowing the path and walking the path; it is of utmost importance to roll up your sleeves and get your hands dirty with kernel source code. Finally, with your thus gained kernel *knowledge*, wherever you go, you will *shine*.

Video Lectures on Kernel Recipes

```
linux-0.11
├── boot
│   ├── bootsect.s      head.s      setup.s
├── fs
│   ├── bitmap.c    block_dev.c buffer.c        char_dev.c  exec.c
│   ├── fcntl.c     file_dev.c  file_table.c    inode.c     ioctl.c
│   ├── namei.c     open.c      pipe.c          read_write.c
│   ├── stat.c      super.c     truncate.c
├── include
│   ├── a.out.h     const.h     ctype.h     errno.h     fcntl.h
│   ├── signal.h    stdarg.h    stddef.h    string.h    termios.h
│   ├── time.h      unistd.h    utime.h
│   ├── asm
│   │   ├── io.h    memory.h    segment.h   system.h
│   ├── linux
│   │   ├── config.h    fdreg.h fs.h    hdreg.h     head.h
│   │   ├── kernel.h    mm.h    sched.h sys.h       tty.h
│   ├── sys
│   │   ├── stat.h      times.h types.h utsname.h   wait.h
├── init
│   └── main.c
├── kernel
│   ├── asm.s       exit.c      fork.c      mktime.c    panic.c
│   ├── printk.c    sched.c     signal.c    sys.c       system_calls.s
│   ├── traps.c     vsprintf.c
│   ├── blk_drv
│   │   ├── blk.h   floppy.c    hd.c    ll_rw_blk.c     ramdisk.c
│   ├── chr_drv
│   │   ├── console.c   keyboard.S  rs_io.s
│   │   ├── serial.c    tty_io.c    tty_ioctl.c
│   ├── math
│   │   ├── math_emulate.c
├── lib
│   ├── close.c  ctype.c  dup.c     errno.c  execve.c  _exit.c
│   ├── malloc.c open.c   setsid.c  string.c wait.c    write.c
├── Makefile
├── mm
│   ├── memory.c page.s
└── tools
    └── build.c
```

1. Beginner's start with Linux 0.11 source (less than 20,000 lines of source code). After 20 years of development, compared with Linux 0.11, Linux has become very huge, complex, and difficult to learn. But the design concept and main structure have no fundamental changes. Learning Linux 0.11 still has important practical significance.
2. Mandatory Reading for Kernel Hackers => `Linux_source_dir/Documentation/*`
3. You should be subscribed and active on at-least one kernel mailing list. Start with kernel newbies.
4. You do not need to read the full source code. Once you are familiar with the kernel API's and its usage, directly start with the source code of the sub-system you are interested in. You can also start with writing your own plug-n-play modules to experiment with the kernel.
5. Device Driver writers would benefit by having their own dedicated hardware. Start with Raspberry Pi.

edited Jul 1 at 4:21

answered Jan 5 at 15:03

manav m-n
2,968 6 34 70

1 Good to see a lasting, comprehensive and non-link dependent answer on a question like this. +1 – Rudi Kershaw Jan 8 at 9:01

I hope to be artsy enough to understand the beauty of this answer one day! – Hamzahfrq Jun 8 at 13:05

@manav m-n: May you say why you post `Solovay-Strassen primality test` in your answer. I think you want to express something, but I didn't get it. – user565739 Jul 2 at 6:21

A hacker cannot, as they devastatingly put it “approach problem-solving like a plumber in a hardware store”; you have to know what the components actually do. This code is put as an example to the following paragraph... to weigh and consider... – manav m-n Jul 2 at 18:54

Check out The Linux Kernel Janitor Project

> "We go through the Linux kernel source code, doing code reviews, fixing up unmaintained code and doing other cleanups and API conversions. It is a good start to kernel hacking."

edited Jan 5 at 12:48

Michaël
2,626 7 25 50

answered May 27 '09 at 9:35

John Smith
2,536 2 19 30

1 +1 awesome link – Ethan Heilman Aug 13 '09 at 20:10

1 For those interested in learning more about The Linux Kernel Janitor Project, I wrote my master thesis analysing their work. That report is now recently published as a book, ISBN 978-3-639-20637-1. bookfinder.com/search/… – hlovdal Nov 24 '09 at 0:27

I would have to say:"learn C". :)

Try this free online book.

Linux Kernel Module Programming Guide
http://www.linuxhq.com/guides/LKMPG/mpg.html

answered May 27 '09 at 8:59

Antonio Louro
505 1 5 13

Following books I read and found very useful :

1. Linux Kernel Development
2. Understanding the Linux Kernel, Third Edition
3. Professional Linux Kernel Architecture
4. Essential Linux Device Drivers
5. Linux Device Drivers
6. Linux Kernel Programming (3rd Edition)
7. LF320 Linux Kernel Internals and Debugging

8. Linux System Programming: Talking Directly to the Kernel and C Library
9. Writing Linux Device Drivers: a guide with exercises

edited Jan 11 at 7:43

Abhijeet Kasurde
648 7 14

answered May 18 '12 at 19:11

Raulp
1,608 1 24 48

Check kernelnewbies.org, subscribe to the Kernelnewbies mailing list, got to irc.oftc.org #kernelnewbies

answered May 27 '09 at 11:16
chill

Some resources:

- Book: Robert Love, Linux Kernel Development, 3rd edition
- Book: Corbet, Rubini, Linux Device Drivers, 3rd edition (free version here)
- Book: Linux kernel in a nutshell (free version here)
- Free material provided by Free Electrons

answered Oct 31 '13 at 11:56

Claudio
3,645 4 27
