# LEARNING linux-kernel

#linux-kernel

# Table of Contents

## About

# Chapter 1: Getting started with linux-kernel

## Remarks

This section provides an overview of what linux-kernel is, and why a developer might want to use it.

It should also mention any large subjects within linux-kernel, and link out to the related topics. Since the Documentation for linux-kernel is new, you may need to create initial versions of those related topics.

## Versions

| Version | Release date |
|---|---|
| 4.4 | 2016-01-10 |
| 4.1 | 2015-06-21 |
| 3.18 | 2014-12-07 |
| 3.16 | 2014-08-03 |
| 3.12 | 2013-11-03 |
| 3.10 | 2013-06-30 |
| 3.4 | 2012-05-20 |
| 3.2 | 2012-01-04 |

## Examples

### Installation or Setup

Linux kernel source code can be found in https://www.kernel.org/

# Download extract and enter to the kernel directory

Type these commands step by steps in your terminal.(Choose the appropriate version you needed instead of linux-4.7.tar.gz )

```
wget http://www.kernel.org/pub/linux/kernel/v4.7/linux-4.7.tar.gz
tar zxvf linux-4.7.tar.gz
cd linux-4.7
```

`make menuconfig` will select the features required for the kernel. Old kernel configurations can be copied by using old `.config` file and executing `make oldconfig`. Also we can use `make xconfig` as a graphical version of the configuration tool.

## Build the dependencies, compile the kernel and modules.

```
make dep
make bzImage
make modules
make modules_install
```

Alternatively if you want to reconfigure the old kernel and re compile it, execute the below commands:

```
make mrproper
make menuconfig
make dep
make clean
make bzImage
make modules
make modules_install
```

Then copy the kernel, `system.map` file to `/boot/vmlinuz-4.7`

create a `.conf` file with the below content

```
image = /boot/vmlinuz-4.7
label = "Linux 4.7"
```

Then execute `lilo -v` to modify the boot sector and reboot.

Read Getting started with linux-kernel online: https://riptutorial.com/linux-kernel/topic/2385/getting-started-with-linux-kernel

# Chapter 2: Creation and usage of Kernel Threads

## Introduction

This topic discusses how to create and use kernel threads.

## Examples

### Creation of kernel threads

kern_thread.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/kthread.h>
#include <linux/sched.h>

#define AUTHOR        "Nachiket Kulkarni"
#define DESCRIPTION    "Simple module that demonstrates creation of 2 kernel threads"

static int kthread_func(void *arg)
{
/* Every kthread has a struct task_struct associated with it which is it's identifier.
* Whenever a thread is schedule for execution, the kernel sets "current" pointer to
* it's struct task_struct.
* current->comm is the name of the command that caused creation of this thread
* current->pid is the process of currently executing thread
*/
    printk(KERN_INFO "I am thread: %s[PID = %d]\n", current->comm, current->pid);
    return 0;
}

static int __init init_func(void)
{
    struct task_struct *ts1;
    struct task_struct *ts2;
    int err;

    printk(KERN_INFO "Starting 2 threads\n");

/*struct task_struct *kthread_create(int (*threadfn)(void *data), void *data, \
 *                         const char *namefmt, ...);
 * This function creates a kernel thread and starts the thread.
 */
    ts1 = kthread_run(kthread_func, NULL, "thread-1");
    if (IS_ERR(ts1)) {
        printk(KERN_INFO "ERROR: Cannot create thread ts1\n");
        err = PTR_ERR(ts1);
        ts1 = NULL;
        return err;
    }
    ts1 = kthread_run(kthread_func, NULL, "thread-1");
    if (IS_ERR(ts1)) {
        printk(KERN_INFO "ERROR: Cannot create thread ts1\n");
        err = PTR_ERR(ts1);
        ts1 = NULL;
        return err;
    }

    printk(KERN_INFO "I am thread: %s[PID = %d]\n", current->comm, current->pid);
    return 0;
}

static void __exit exit_func(void)
{
    printk(KERN_INFO "Exiting the module\n");
}

module_init(init_func);
module_exit(exit_func);

MODULE_AUTHOR(AUTHOR);
MODULE_DESCRIPTION(MODULE_AUTHOR);
MODULE_LICENSE("GPL");
```

Makefile

```
obj-m += kern_thread.o

all:
    make -C /lib/module/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/module/$(shell uname -r)/build M=$(PWD) clean
```

Upon inserting the .ko, it printed:

```
Starting 2 threads
I am thread: thread-1[PID = 6786]
I am thread: insmod[PID = 6785]
I am thread: thread-2[PID = 6788]
```

Read Creation and usage of Kernel Threads online: https://riptutorial.com/linux-kernel/topic/10619/creation-and-usage-of-kernel-threads

# Chapter 3: Event Tracing

## Examples

### Tracing I2C Events

Note: I am assuming that `debugfs` is mounted under `/sys/kernel/debug`

If not, try:

```
mount -t debugfs none /sys/kernel/debug
```

Change into the tracing directory:

```
cd /sys/kernel/debug/tracing/
```

Make sure the function tracer is disabled:

```
echo nop > current_tracer
```

Enable all I2C events:

```
echo 1 > events/i2c/enable
```

Make sure tracing is enabled:

```
echo 1 > tracing_on
```

The trace messages can be viewed in `/sys/kernel/debug/tracing/trace`, example:

```
... i2c_write: i2c-5 #0 a=044 f=0000 l=2 [02-14]
... i2c_read: i2c-5 #1 a=044 f=0001 l=4
... i2c_reply: i2c-5 #1 a=044 f=0001 l=4 [33-00-00-00]
... i2c_result: i2c-5 n=2 ret=2
```

The trace events user-space API documentation can be found in the file `Documentation/trace/events.txt` of the kernel source.

Read Event Tracing online: https://riptutorial.com/linux-kernel/topic/3466/event-tracing

# Chapter 4: Fork System call

## Examples

### fork() system call

`fork()` is a system call. fork is used to create a child process from the running process, which is a replica of the parent process(Process which executed `fork()` ). Child process is derived from the parent process. Both the parent and child have different address space, each is independent of the changes made to the variables.

The child process has its own PID(process identification). PPID(Parent Process ID) of child process is same as PID of parent process.

> Format:
>
> Header file : `#include <unistd.h>`
> Function Declaration : `pid_t fork(void);`

fork() doesn't need any input arguments.

On successful creation of child process the pid of the child process is returned to the parent process and 0 is returned in the child process. On Failure return `-1` with no process created.

Usage example:

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

void child_process();
void parent_process();

int main()
{
    pid_t pid;
    pid=fork();
    if(pid==0)
        child_process();
    else
        parent_process();
    return 0;
}

/*getpid() will return the Pid of the
  current process executing the function */

void child_process()
{
    printf("Child process with PID : %d  and PPID : %d ", getpid(),getppid());
}

void parent_process()
```

```
{
    printf("Parent process with PID : %d", getpid());
}
```

The sequence of the `printf` statements from the child and parent depend on the scheduling mechanism which purely depends on the system.

Read Fork System call online: https://riptutorial.com/linux-kernel/topic/5199/fork-system-call

# Chapter 5: How to find the right person for help.

## Introduction

This should mirror some of the official Linux kernel docs, and post links to the latest versions of said documents in `tovalds/linux` on GitHub.com. The idea is to encourage individuals to make use of the `MAINTAINERS` files, `linux-kernel` mailing list, `git log`, and `scripts/get-maintainer`, so that they are familiar with the commonly-used ways of identifying a key point of contact.

## Examples

### Find the "likely" maintainers for the FTDI USB serial converter

First, determine the source file for this particular driver. Found it at `drivers/usb/serial/ftdi_sio.c`.

```
./scripts/get_maintainer.pl drivers/usb/serial/ftdi_sio.c
```

And the results:

```
Johan Hovold <johan@kernel.org> (maintainer:USB SERIAL SUBSYSTEM)
Greg Kroah-Hartman <gregkh@linuxfoundation.org> (supporter:USB SUBSYSTEM)
linux-usb@vger.kernel.org (open list:USB SERIAL SUBSYSTEM)
linux-kernel@vger.kernel.org (open list)
```

Now we know who to ping for help with this particular driver, and which e-mail addresses should be CC'ed when submitting a patch against this driver.

Read How to find the right person for help. online: https://riptutorial.com/linux-kernel/topic/10056/how-to-find-the-right-person-for-help-

# Chapter 6: Linux Hello World Device driver

## Examples

### An empty kernel module

```
#include <linux/init.h>
#include <linux/module.h>

/**
 * This function is called when the module is first loaded.
 */
static int __init hello_kernel_init(void)
{
    printk("Hello, World!\n");
    return 0;
}

/**
 * This function is called when is called if and when the module is unloaded.
 */
static void __exit hello_kernel_exit(void)
{
    printk("Goodbye, cruel world...\n");
}

/* The names of the init/exit functions are arbitrary, and they are bound using the following
macro definitions */
module_init(hello_kernel_init);
module_exit(hello_kernel_exit);
```

In order to write a Linux device driver (Character-device, Block-device, etc...), it is necessary to create a kernel module that has an entry and exit points.

By itself, the kernel module does nothing; it has no meaningful way to communicate with the userspace. Using the entry point it is possible to create a new character-device, for example, which is then used to communicate with the userspace.

### Building and running the module

In order to compile the driver, it is necessary to have the Linux Kernel source tree.

Assuming the sources are at `/lib/modules/<kernel-version>`, the following Makefile will compile the file `driver.c` into the `driver.ko` Kernel Object

```
obj-m := driver.o
KDIR := /lib/modules/$(shell uname -r)/build/
PWD := $(shell pwd)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
```

Notice how this Makefile calls `make` in the build directory of the Kernel.

When the compilation step finishes successfully, the src directory of the driver will look somewhat like this:

```
driver.c  driver.ko  driver.mod.c  driver.mod.o  driver.o  Makefile  modules.order
Module.symvers
```

In order to "run" the module, it is necessary to insert into the running kernel:

```
$ insmod driver.ko
$ dmesg | tail -n 1
[133790.762185] Hello, World!

$ rmmod driver.ko
$ dmesg | tail -n 1
[133790.762185] Goodbye, cruel world...
```

Read Linux Hello World Device driver online: https://riptutorial.com/linux-kernel/topic/7056/linux-hello-world-device-driver

# Chapter 7: Linux: Named Pipes(FIFO)

## Examples

### What is Named Pipe (FIFO)

```
A named pipe is really just a special kind of file (a FIFO file) on the local hard drive.
Unlike a regular file, a FIFO file does not contain any user information. Instead, it allows
two or more processes to communicate with each other by reading/writing to/from this file.

A named pipe works much like a regular pipe, but does have some noticeable differences.

Named pipes exist as a device special file in the file system.
Processes of different ancestry can share data through a named pipe.
When all I/O is done by sharing processes, the named pipe remains in the file system for later
use.

The easiest way to create a FIFO file is to use the mkfifo command. This command is part of
the standard Linux utilities and can simply be typed at the command prompt of your shell. You
may also use the mknod command to accomplish the same thing.
```

Read Linux: Named Pipes(FIFO) online: https://riptutorial.com/linux-kernel/topic/6144/linux--named-pipes-fifo-

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with linux-kernel | Community, EsmaeelE, Marek Skiba, Matt, Tejus Prasad, vinay hunachyal |
| 2 | Creation and usage of Kernel Threads | nachiketkulk |
| 3 | Event Tracing | sergej |
| 4 | Fork System call | chait, Dilip Kumar |
| 5 | How to find the right person for help. | DevNull, EsmaeelE |
| 6 | Linux Hello World Device driver | Gilad Naaman |
| 7 | Linux: Named Pipes(FIFO) | chait |