Бесплатная электронная книга

учусь

linux-kernel

Free unaffiliated eBook created from
Stack Overflow contributors.

#linux-

kernel

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: linux-kernel

It is an unofficial and free linux-kernel ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official linux-kernel.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с linux-kernel

замечания

В этом разделе представлен обзор того, что такое linux-ядро, и почему разработчик может захотеть его использовать.

Следует также упомянуть о любых крупных объектах в Linux-ядре и ссылаться на связанные темы. Поскольку документация для linux-ядра является новой, вам может потребоваться создать начальные версии этих связанных тем.

Версии

Версия	Дата выхода
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

Установка или настройка

Исходный код ядра Linux можно найти в https://www.kernel.org/

Загрузите извлечение и введите в каталог ядра

Введите эти команды шаг за шагом в своем терминале. (Выберите нужную вам версию

вместо 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` выберет функции, необходимые для ядра. Старые конфигурации ядра могут быть скопированы с использованием старого файла `.config` и выполнения `make oldconfig` . Также мы можем использовать `make xconfig` как графическую версию инструмента конфигурации.

Создайте зависимости, скомпилируйте ядро и модули.

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

Альтернативно, если вы хотите перенастроить старое ядро и выполнить его компиляцию, выполните следующие команды:

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

Затем скопируйте ядро, `system.map` файл `/boot/vmlinuz-4.7`

создайте файл `.conf` с содержимым ниже

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

Затем выполните `lilo -v` для изменения загрузочного сектора и перезагрузки.

Прочитайте Начало работы с linux-kernel онлайн: https://riptutorial.com/ru/linux-kernel/topic/2385/начало-работы-с-linux-kernel

глава 2: Linux: Именованные каналы (FIFO)

Examples

Что такое именованная труба (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.
```

Прочитайте Linux: Именованные каналы (FIFO) онлайн: https://riptutorial.com/ru/linux-kernel/topic/6144/linux--именованные-каналы--fifo-

глава 3: Драйвер устройства Hello Hello World

Examples

Пустой модуль ядра

```
#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);
```

Чтобы написать драйвер устройства Linux (Character-device, Block-device и т. Д.), Необходимо создать модуль ядра, который имеет точки входа и выхода.

Сам по себе модуль ядра ничего не делает; он не имеет значимого способа общения с пользовательским пространством. Используя точку входа, можно создать, например, новое символьное устройство, которое затем используется для связи с пользовательским пространством.

Создание и запуск модуля

Чтобы скомпилировать драйвер, необходимо иметь исходное дерево ядра ядра Linux.

Предполагая, что исходные файлы находятся в `/lib/modules/<kernel-version>` , следующий файл Makefile будет скомпилировать файл `driver.c` в файл `driver.ko` Kernel

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

```
PWD := $(shell pwd)

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

Обратите внимание на то, как это Makefile вызовы `make` в каталог сборки ядра.

Когда этап компиляции завершится успешно, каталог src драйвера будет выглядеть примерно так:

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

Чтобы «запустить» модуль, необходимо вставить в запущенное ядро:

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

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

Прочитайте Драйвер устройства Hello Hello World онлайн: https://riptutorial.com/ru/linux-kernel/topic/7056/драйвер-устройства-hello-hello-world

глава 4: Как найти нужного человека для помощи.

Вступление

Это должно отражать некоторые официальные документы ядра Linux и размещать ссылки на последние версии указанных документов в `tovalds/linux` на GitHub.com. Идея состоит в том, чтобы побудить людей использовать файлы `MAINTAINERS` , список рассылки `linux-kernel` , `git log` и `scripts/get-maintainer` , чтобы они были знакомы с обычно используемыми способами идентификации ключевого пункта контакта.

Examples

Найдите «вероятных» сопровождающих устройств для последовательного преобразователя USB FTDI

Сначала определите исходный файл для этого конкретного драйвера. Найденный в `drivers/usb/serial/ftdi_sio.c` .

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

И результаты:

```
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)
```

Теперь мы знаем, кто пингует за помощью с этим конкретным драйвером и какие адреса электронной почты должны быть указаны при отправке патча против этого драйвера.

Прочитайте Как найти нужного человека для помощи. онлайн: https://riptutorial.com/ru/linux-kernel/topic/10056/как-найти-нужного-человека-для-помощи-

глава 5: Системный вызов Fork

Examples

системный вызов fork ()

`fork()` - системный вызов. fork используется для создания дочернего процесса из текущего процесса, который является репликой родительского процесса (Process, который выполнял `fork()`). Детский процесс происходит из родительского процесса. И родительский, и дочерний имеют другое адресное пространство, каждое из которых не зависит от изменений, внесенных в переменные.

У дочернего процесса есть собственный PID (идентификация процесса). Идентификатор PPID (идентификатор родительского процесса) дочернего процесса такой же, как PID родительского процесса.

> Формат:
>
> Заголовочный файл: `#include <unistd.h>`
> Объявление функции: `pid_t fork(void);`

fork () не нуждается ни в каких входных аргументах.

При успешном создании дочернего процесса pid дочернего процесса возвращается родительскому процессу, а 0 возвращается в дочерний процесс. On Failure return `-1` без создания процесса.

Пример использования:

```
#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());
}
```

Последовательность инструкций `printf` от дочернего элемента и родителя зависит от механизма планирования, который зависит исключительно от системы.

Прочитайте Системный вызов Fork онлайн: https://riptutorial.com/ru/linux-kernel/topic/5199/системный-вызов-fork

глава 6: Создание и использование потоков ядра

Вступление

В этом разделе обсуждается, как создавать и использовать потоки ядра.

Examples

Создание потоков ядра

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
```

После вставки .ko, он печатается:

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

Прочитайте Создание и использование потоков ядра онлайн: https://riptutorial.com/ru/linux-kernel/topic/10619/создание-и-использование-потоков-ядра

глава 7: Трассировка событий

Examples

Трассировка событий I2C

Примечание. Я предполагаю, что `debugfs` монтируется под `/sys/kernel/debug`

Если нет, попробуйте:

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

Перейдите в каталог трассировки:

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

Убедитесь, что функция трассировщика отключена:

```
echo nop > current_tracer
```

Включить все события I2C:

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

Убедитесь, что трассировка включена:

```
echo 1 > tracing_on
```

Сообщения трассировки можно просмотреть в `/sys/kernel/debug/tracing/trace` , например:

```
... 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
```

Информацию о API-интерфейсе пользовательского пространства трассировки можно найти в файле `Documentation/trace/events.txt` источника ядра.

Прочитайте Трассировка событий онлайн: https://riptutorial.com/ru/linux-kernel/topic/3466/трассировка-событий

кредиты

S. No	Главы	Contributors
1	Начало работы с linux-kernel	Community, EsmaeelE, Marek Skiba, Matt, Tejus Prasad, vinay hunachyal
2	Linux: Именованные каналы (FIFO)	chait
3	Драйвер устройства Hello Hello World	Gilad Naaman
4	Как найти нужного человека для помощи.	DevNull, EsmaeelE
5	Системный вызов Fork	chait, Dilip Kumar
6	Создание и использование потоков ядра	nachiketkulk
7	Трассировка событий	sergej