eBook Gratuit

APPRENEZ linux-kernel

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#linux-kernel

Table des matières

À propos

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

Chapitre 1: Démarrer avec linux-kernel

Remarques

Cette section fournit une vue d'ensemble de ce qu'est linux-kernel et pourquoi un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les grands sujets dans linux-kernel, et établir un lien vers les rubriques connexes. La documentation de linux-kernel étant nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Version	Date de sortie
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 ou configuration

Le code source du noyau Linux peut être trouvé dans https://www.kernel.org/

Télécharger l'extrait et entrer dans le répertoire du noyau

Tapez ces commandes pas à pas dans votre terminal (choisissez la version appropriée à la place de 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` sélectionne les fonctionnalités requises pour le noyau. Les anciennes configurations du noyau peuvent être copiées en utilisant l'ancien fichier `.config` et en exécutant `make oldconfig` . Nous pouvons également utiliser `make xconfig` comme version graphique de l'outil de configuration.

Construisez les dépendances, compilez le noyau et les modules.

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

Si vous souhaitez reconfigurer l'ancien noyau et le compiler, exécutez les commandes suivantes:

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

Ensuite , copiez le noyau, `system.map` fichier à `/boot/vmlinuz-4.7`

créer un fichier `.conf` avec le contenu ci-dessous

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

Ensuite, exécutez `lilo -v` pour modifier le secteur de démarrage et redémarrer.

Lire Démarrer avec linux-kernel en ligne: https://riptutorial.com/fr/linux-kernel/topic/2385/demarrer-avec-linux-kernel

Chapitre 2: Appel système fourche

Examples

appel système fork ()

`fork()` est un appel système. fork est utilisé pour créer un processus enfant à partir du processus en cours d'exécution, qui est une réplique du processus parent (Processus qui a exécuté `fork()`). Le processus enfant est dérivé du processus parent. Le parent et l'enfant ont des espaces d'adressage différents, chacun étant indépendant des modifications apportées aux variables.

Le processus enfant a son propre PID (identification de processus). PPID (ID de processus parent) du processus enfant est identique à celui du processus parent.

> Format:
>
> Fichier d'en-tête: `#include <unistd.h>`
> Déclaration de fonction: `pid_t fork(void);`

fork () n'a pas besoin d'arguments en entrée.

En cas de création réussie du processus enfant, le pid du processus enfant est renvoyé au processus parent et 0 est renvoyé dans le processus enfant. En cas d'échec, renvoyer `-1` sans processus créé.

Exemple d'utilisation:

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

La séquence des instructions `printf` de l'enfant et du parent dépend du mécanisme de planification qui dépend uniquement du système.

Lire Appel système fourche en ligne: https://riptutorial.com/fr/linux-kernel/topic/5199/appel-systeme-fourche

Chapitre 3: Comment trouver la bonne personne pour vous aider.

Introduction

Cela devrait refléter certains des documents officiels du noyau Linux et publier des liens vers les dernières versions de ces documents dans `tovalds/linux` sur GitHub.com. L'idée est d'encourager les individus à utiliser les fichiers `MAINTAINERS` , la liste de diffusion `linux-kernel` , `git log` et les `scripts/get-maintainer` , afin qu'ils soient familiarisés avec les méthodes couramment utilisées pour identifier un point de contact clé.

Examples

Trouvez les responsables "probables" du convertisseur série USB FTDI

Tout d'abord, déterminez le fichier source de ce pilote particulier. `drivers/usb/serial/ftdi_sio.c` trouvé dans `drivers/usb/serial/ftdi_sio.c` .

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

Et les résultats:

```
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)
```

Nous savons maintenant à qui demander de l'aide avec ce pilote particulier et quelles adresses de messagerie doivent être envoyées lors de l'envoi d'un correctif à ce pilote.

Lire Comment trouver la bonne personne pour vous aider. en ligne: https://riptutorial.com/fr/linux-kernel/topic/10056/comment-trouver-la-bonne-personne-pour-vous-aider-

Chapitre 4: Création et utilisation des threads du noyau

Introduction

Cette rubrique explique comment créer et utiliser des threads du noyau.

Examples

Création de threads du noyau

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
```

En insérant le .ko, il imprimait:

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

Lire Création et utilisation des threads du noyau en ligne: https://riptutorial.com/fr/linux-kernel/topic/10619/creation-et-utilisation-des-threads-du-noyau

Chapitre 5: Linux: Named Pipes (FIFO)

Examples

Qu'est-ce que le tube nommé (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.
```

Lire Linux: Named Pipes (FIFO) en ligne: https://riptutorial.com/fr/linux-kernel/topic/6144/linux--named-pipes--fifo-

Chapitre 6: Pilote Linux Hello World Device

Examples

Un module noyau vide

```
#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);
```

Pour écrire un pilote de périphérique Linux (Character-device, Block-device, etc.), il est nécessaire de créer un module de noyau comportant des points d'entrée et de sortie.

En soi, le module du noyau ne fait rien; il n'a aucun moyen significatif de communiquer avec l'espace utilisateur. En utilisant le point d'entrée, il est possible de créer un nouveau périphérique de caractère, par exemple, qui est ensuite utilisé pour communiquer avec l'espace utilisateur.

Construire et exécuter le module

Pour compiler le pilote, il est nécessaire d'avoir l'arbre source du noyau Linux.

En supposant que les sources se trouvent dans `/lib/modules/<kernel-version>` , le fichier Makefile suivant compilera le fichier `driver.c` dans l' `driver.ko` noyau `driver.ko`

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

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

Remarquez comment ces appels Makefile sont `make` dans le répertoire de compilation du noyau.

Lorsque l'étape de compilation se termine avec succès, le répertoire src du pilote ressemblera à ceci:

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

Pour "exécuter" le module, il est nécessaire d'insérer dans le noyau en cours d'exécution:

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

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

Lire Pilote Linux Hello World Device en ligne: https://riptutorial.com/fr/linux-kernel/topic/7056/pilote-linux-hello-world-device

Chapitre 7: Suivi des événements

Examples

Suivi des événements I2C

Note: Je suppose que `debugfs` est monté sous `/sys/kernel/debug`

Sinon, essayez:

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

Accédez au répertoire de suivi:

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

Assurez-vous que le traceur de fonction est désactivé:

```
echo nop > current_tracer
```

Activer tous les événements I2C:

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

Assurez-vous que le suivi est activé:

```
echo 1 > tracing_on
```

Les messages de trace peuvent être visualisés dans `/sys/kernel/debug/tracing/trace` , exemple:

```
... 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
```

La documentation de l'API de l'espace utilisateur des événements de trace se trouve dans le fichier `Documentation/trace/events.txt` de la source du noyau.

Lire Suivi des événements en ligne: https://riptutorial.com/fr/linux-kernel/topic/3466/suivi-des-evenements

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec linux-kernel	Community, EsmaeelE, Marek Skiba, Matt, Tejus Prasad, vinay hunachyal
2	Appel système fourche	chait, Dilip Kumar
3	Comment trouver la bonne personne pour vous aider.	DevNull, EsmaeelE
4	Création et utilisation des threads du noyau	nachiketkulk
5	Linux: Named Pipes (FIFO)	chait
6	Pilote Linux Hello World Device	Gilad Naaman
7	Suivi des événements	sergej