OS2 Drivers Linux

Licence

La première partie de cette présentation est dérivée de:

Embedded Linux kernel and driver development

Traduction française par Julien Boibessot

La seconde partie de cette présentation est dérivée de:

Linux USB drivers

Contenu du cours (1)

Développement de pilotes

- Pilotes de périphériques Linux
- Un simple module
- Contraintes de programmation
- (Dé)chargement des modules
- Paramètres des modules
- Dépendances des modules
- Ajouter des sources à l'arborescence du noyau
- Déboguage du noyau

Contenu du cours (2)

Développement avancé de pilotes

- Gestion mémoire
- Registres I/O et accès mémoire
- Pilotes de périphériques caractères
- Endormissement, interruptions
- mmap, DMA
- Nouveau modèle de périphérique, sysfs
- Hotplug
- Fichier de périphériques dynamiques avec udev

Contenu du cours (3)

Linux USB Drivers

- Linux USB basics
  - Linux USB drivers
  - USB devices
  - User-space representation
  - Linux USB communication
- USB Request Blocks
  - Initializing and submitting URBs
  - Completion handlers
- Writing USB drivers
  - Supported devices
  - Registering a USB driver
  - USB transfers without URBs

Drivers Linux

Développement de pilotes de périphériques

Développement de pilotes de périphériques

Pilotes de périphériques pour Linux

Pilotes caractère

- Communication grâce à un flux séquentiel de caractères individuels
- Les pilotes caractère peuvent être identifiés par leur type `c` (`ls -l`):

```
crw-rw----  1 root uucp 4, 64 Feb 23  2004 /dev/ttyS0
crw--w----  1 jdoe tty 136, 1 Sep 13 06:51 /dev/pts/1
crw-------  1 root root 13, 32 Feb 23
2004 /dev/input/mouse0
crw-rw-rw-  1 root root 1, 3 Feb 23  2004 /dev/null
```

- Exemples: clavier, souris, port parallèle, IrDA, Bluetooth, consoles, terminaux...

Pilotes bloc

- Accès par blocs de données de taille fixe. On peut accéder aux blocs dans n'importe quel ordre.
- Les pilotes blocs peuvent être identifiés par leur type `b` (`ls -l`):

```
brw-rw----   1 root disk 3, 1 Feb 23  2004 /dev/hda1
brw-rw----   1 jdoe floppy   2,   0 Feb 23  2004 fd0
brw-rw----   1 root disk     7,   0 Feb 23  2004 loop0
brw-rw----   1 root disk     1,   1 Feb 23  2004 ram1
brw-------   1 root root     8,   1 Feb 23  2004 sda1
```

- Exemples: disques durs, disques mémoires, périphériques de loopback (images de systèmes de fichiers)...

Nombres majeurs et nombres mineurs

Comme vous pouvez le voir dans l'exemple précédent, les périphériques ont 2 numéros qui leurs sont associés:

- Premier numéro: nombre majeur
  Associé de manière unique à chaque pilote
- Second numéro: nombre mineur
  Associé de manière unique à chaque périphérique / entrée dans /dev

Pour trouver quel pilote correspond à un périphérique, regardez `Documentation/devices.txt`

Création des fichiers de périphériques

- Les fichiers de périphériques ne sont pas créés (par défaut) lorsqu'un pilote est chargé.
- Ils doivent être créés par avance:

```
mknod /dev/<device> [c|b] <major> <minor>
```

- Exemples:

```
mknod /dev/ttyS0 c 4 64
mknod /dev/hda1  b 3 1
```

Autres types de pilotes

Ils n'ont aucune entrée correspondante dans `/dev` dans laquelle vous pouvez lire ou écrire avec une commande Unix standard.

- Les pilotes réseaux
  Ils sont représentés par un périphérique réseau comme `ppp0`, `eth1`, `usbnet`, `irda0` (liste: `ifconfig -a`)
- Les autres pilotes
  Souvent, ce sont des pilotes intermédiaires servant d'interface avec d'autres.

Développement de pilotes

Un exemple de module

Le module hello

Merci à Jonathan Corbet pour l'exemple !

```
/* hello.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
```

License des modules

La liste des licences est détaillée dans `include/linux/module.h`

- GPL
  GNU Public License v2 ou supérieure
- GPL v2
  GNU Public License v2
- GPL and additional rights
- Dual BSD/GPL
  Choix entre GNU Public License v2 et BSD
- Dual MPL/GPL
  Choix entre GNU Public License v2 et Mozilla
- Propriétaire
  Produits non libres

Utilité des licences de module

- Utilisées par les développeurs du noyau pour identifier des problèmes venant de pilotes propriétaires, qu'ils n'essaierons pas de résoudre
- Permettent aux utilisateurs de vérifier que leur système est à 100% libre
- Permettent aux distributeurs GNU/Linux de vérifier la conformité à leur politique de licence

Règles de codage des modules (1)

- Includes C: vous ne pouvez pas utiliser les fonctions de la bibliothèque C standard (`printf()`, `strcat()`, etc.). La bibliothèque C est implémentée au dessus du noyau et non l'inverse.
- Linux a quelques fonctions C utiles comme `printk()`, qui possède une interface similaire à `printf()`.

Donc, seul les fichiers d'entêtes du noyau sont autorisés.

Règles de codage des modules (2)

- N'utilisez jamais de nombres à virgule flottante dans le code du noyau. Votre code peut être exécuté sur un processeur sans unité de calcul à virgule flottante (comme sur ARM). L'émulation par le noyau est possible mais très lente.
- Définissez tous vos symboles en local/statique, hormis ceux qui sont exportés (afin d'éviter la pollution de l'espace de nommage).
- Consultez: `Documentation/CodingStyle`
- Il est toujours bon de connaître, voir d'appliquer, les règles de codage GNU: http://www.gnu.org/prep/standards.html

Compiler un module

- Le Makefile ci-dessous est réutilisable pour tout module Linux 2.6.
- Lancez juste `make` pour construire le fichier hello.ko
- Attention: assurez vous qu'il y ait une `[Tabulation]` au début de la ligne `$(MAKE)` (syntaxe requise par make)

```
# Makefile for the hello module

obj-m := hello.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Tabulation
(pas d'espaces)

Utilisation du module

- En tant que root, lancez
  `tail -f /var/log/messages`
- En tant que root dans un autre terminal, chargez le module:
  `insmod ./hello.ko`
- Vous verrez ce message dans `/var/log/messages`:
  `Sep 13 22:02:30 localhost kernel: Hello, world`
- Maintenant déchargez le module:
  `rmmod hello`
- Vous verrez:
  `Sep 13 22:02:37 localhost kernel: Goodbye, cruel world`

Utilitaires pour les modules

- `insmod <nom_du_module>`
  `insmod <chemin_du_module>.ko`
  Essaie de charger le module donné, si nécessaire en cherchant le fichier `.ko` dans les répertoires par défaut (qui peuvent être redéfinis par la variable d'environ. `MODPATH`).
- `modprobe <nom_du_module>`
  Usage le plus courant pour charger un module: essaie de charger tous les modules dont dépend le module donné, puis le module lui-même. Plusieurs options sont disponibles.
- `rmmod <nom_du_module>`
  Essaie de décharger le module donné.

Développement de pilotes

Définir et passer des paramètres aux modules

Le module hello avec des paramètres

```
/* hello_param.c */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>

MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in: how many times we say
   hello, and to whom */

static char *whom = "world";
module_param(whom, charp, 0);
static int howmany = 1;

module_param(howmany, int, 0);

static int hello_init(void)
{
    int i;
    for (i = 0; i < howmany; i++)
      printk(KERN_ALERT "(%d) Hello, %s\n", i, whom);
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel %s\n", whom);
}

module_init(hello_init);
module_exit(hello_exit);
```

Merci à Jonathan Corbet pour l'exemple !

Utiliser le module hello_param

- Charger le module. Par exemple:
  `insmod ./hello_param.ko howmany=2 whom=universe`
- Vous verrez cela dans `/var/log/messages`:
```
Sep 13 23:04:30 localhost kernel: (0) Hello, universe
Sep 13 23:04:30 localhost kernel: (1) Hello, universe
```
- Décharger le module:
  `rmmod hello_param`
- Vous verrez:
```
Sep 13 23:04:38 localhost kernel: Goodbye, cruel universe
```

Déclarer des paramètres de module (1)

- `module_param(nom, type, perm);`

  `nom`: nom du paramètre

  `type`: soit `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` ou `invbool` (vérifié à la compilation !)

  `perm`: permissions pour l'entrée correspondante

  dans `/sys/module/<module_name>/<param>`. On peut utiliser `0`.

- `module_param_named(nom, valeur, type, perm);`

  Rend la variable `nom` disponible à l'extérieur du module et lui affecte `valeur` à l'intérieur.

Déclarer des paramètres de module (2)

- `module_param_string(nom, chaine, taille, perm);`

  Crée une variable nom de type charp, pré-remplie avec la chaîne de longueur taille, typiquement sizeof(string)

- `module_param_array(name, type, num, perm);`

  Pour déclarer un tableau de paramètres

Passer des paramètres aux modules

- Avec insmod ou modprobe:

```
insmod ./hello_param.ko howmany=2
  whom=universe
```

- Avec modprobe en changeant le fichier `/etc/modprobe.conf`:

```
options hello_param howmany=2 whom=universe
```

- Avec la ligne de commande du noyau, lorsque le module est lié statiquement au noyau:

```
options hello_param.howmany=2 \
  hello_param.whom=universe
```

Développement de pilotes

Dépendance de modules

Dépendances de modules

- Les dépendances des modules n'ont pas à être spécifiées explicitement par le créateur du module.
- Elles sont déduites automatiquement lors de la compilation du noyau, grâce aux symboles exportés par le module: `module2` dépend de `module1` si `module2` utilise un symbole exporté par `module1`.
- Les dépendances des modules sont stockées dans: `/lib/modules/<version>/modules.dep`
- Ce fichier est mis à jour (en tant que root) avec: `depmod -a [<version>]`

Développement de pilotes

Ajouter des sources au noyau

Nouveau répertoire dans le noyau (1)

Pour ajouter un répertoire `acme_drivers/` aux sources du noyau:

- Déplacer le répertoire `acme_drivers/` à l'endroit approprié dans les sources du noyau
- Créer un fichier `acme_driver/Kconfig`
- Créer un fichier `acme_driver/Makefile` basé sur les variables Kconfig
- Dans le fichier Kconfig du répertoire parent, ajouter: `source “acme_driver/Kconfig”`

Nouveau répertoire dans le noyau (2)

- Dans le fichier Makefile du répertoire parent, ajouter:
  `"obj-$(CONFIG_ACME) += acme_driver/"` (juste 1 condition)
  ou
  `"obj-y += acme_driver/"` (plusieurs conditions)
- Lancer `make xconfig` et utiliser vos nouvelles options !
- Lancer `make` et vos nouveaux fichiers sont compilés !
- Regardez `Documentation/kbuild/*.txt` pour plus de détails

Développement de pilotes

Débogage du noyau

Déboguer avec printk

- Technique universelle de débogage utilisée depuis les débuts de la programmation (observée sur les peintures rupestres)
- Affiché ou non dans la console ou `/var/log/messages` suivant la priorité
- Priorités disponibles (`include/linux/kernel.h`):

```
#define KERN_EMERG    "<0>"  /* système inutilisable */
#define KERN_ALERT    "<1>"  /* une action doit être prise de suite */
#define KERN_CRIT     "<2>"  /* conditions critiques */
#define KERN_ERR      "<3>"  /* conditions d'erreur */
#define KERN_WARNING  "<4>"  /* conditions de warning */
#define KERN_NOTICE   "<5>"  /* condition normale mais significative */
#define KERN_INFO     "<6>"  /* information */
#define KERN_DEBUG    "<7>"  /* messages de débogage */
```

ksymoops

- Aide à décrypter les messages «oops», en convertissant les adresses et le code en informations utiles
- Facile à utiliser: copiez/collez juste le texte oops dans un fichier
- Exemple d'une ligne de commande:

```
ksymoops --no-ksyms -m System.map -v vmlinux
oops.txt
```

- Regardez `Documentation/oops-tracing.txt` et `man ksymoops` pour plus de détails.

Déboguer avec Kprobes

http://www-124.ibm.com/developerworks/oss/linux/projects/kprobes/

- Moyen simple d'insérer des points d'arrêt dans les routines du noyau
- Contrairement au débogage avec printk, vous n'avez pas besoin de recompiler ni de redémarrer votre noyau. Vous avez juste besoin de compiler et charger un module dédié pour déclarer l'adresse de la routine que vous voulez tester.
- Non disruptif, basé sur le gestionnaire d'interruption du noyau
- Kprobes permet même de modifier des registres et des structures de données globales.

Voir http://www-106.ibm.com/developerworks/library/l-kprobes.html pour une présentation

Astuce de débogage du noyau

- Si votre noyau ne démarre pas encore, il est recommandé d'activer le «Low Level debugging» (dans la section «Kernel Hacking» des options du noyau, valable uniquement sur ARM)

```
CONFIG_DEBUG_LL=y
```

Périphériques caractères

Codes d'erreur de Linux

Essayez de reporter les erreurs avec des numéros aussi précis que possible ! Heureusement, les noms de macros sont explicites et vous pouvez vous en rappeler rapidement.

- Codes d'erreur génériques:
  `include/asm-generic/errno-base.h`
- Codes d'erreur spécifiques à une plate-forme:
  `include/asm/errno.h`

Enregistrement des périphériques

- Tout d'abord il faut créer l'(les)entrée(s) correspondante(s) dans `/dev`
- Initialisation du pilote: enregistrement avec un numéro majeur
- Enregistrement (`linux/fs.h`)

```
int register_chrdev(
                unsigned int major,
                const char *name,
                struct file_operations *fops);
```

- Libération du périphérique

```
int unregister_chrdev(
                unsigned int major,
                const char *name);
```

- Si ces fonctions échouent, elles retournent une valeur strictement < 0.

Périphériques enregistrés

> Les périphériques enregistrés sont visibles dans `/proc/devices` avec leur numéro majeur et leur nom:

```
Character devices:
1 mem
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
6 lp
7 vcs
10 misc
13 input
14 sound
...
```

```
Block devices:
1 ramdisk
3 ide0
8 sd
9 md
22 ide1
65 sd
66 sd
67 sd
68 sd
69 sd
...
```

Trouver un numéro majeur libre

- De moins en moins de numéros majeurs sont disponibles
- Il n'est pas recommandé d'en prendre un arbitrairement, car il peut rentrer en conflit avec un autre pilote (standard ou spécifique)
- Solution: laisser `register_chrdev` en trouver un libre dynamiquement pour vous !

```
major = register_chrdev (0, "foo", &name_fops);
```

- Problème: vous ne pouvez pas créer d'entrées `/dev` par avance ! Cependant, le script chargeant le module peut se servir de `/proc/devices`:

```
module=foo; device=foo
insmod $module.ko
major=`awk "\\$2==\"$module\" {print \\$1}" /proc/devices`
mknod /dev/foo0 c $major 0
```

Opérations sur les fichiers (1)

Comme vous venez de le voir, lorsque vous appelez `register_chrdev()`, vous devez déclarer des «file_operations» (appellées *fops*). Voici les principales:

```
> loff_t (*llseek) (struct file *,
                    loff_t, int);

> ssize_t (*read) (struct file *, char *,
                   size_t, loff_t *);

> ssize_t (*write) (struct file *, const char *,
                    size_t, loff_t *);

> int (*open) (struct inode *, struct file *);

> int (*release) (struct inode *, struct file *);
```

Opérations sur les fichiers (2)

> `int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);`

Utilisée pour envoyer au périphérique des commandes spécifiques, qui ne sont ni des lectures, ni des écritures (ex: formater un disque, changer une configuration).

> `int (*mmap) (struct file *, struct vm_area_struct);`

Demande que la mémoire du périphérique soit mappée dans l'espace d'adressage du processus utilisateur

> `struct module *owner;`

Utilisée par le noyau pour garder une trace de qui utilise cette structure et compter le nombre d'utilisateur du module.

La structure « file »

Est créée par le système durant l'appel à `open()`. Représente les fichiers ouverts. Les pointeurs vers cette structure sont les "*fips*".

> `mode_t f_mode;`
>
> Mode d'ouverture du fichier (FMODE_READ, FMODE_WRITE)

> `loff_t f_pos;`
>
> Position dans le fichier ouvert

> `struct file_operations *f_op;`
>
> Peuvent être changées à la volée!

> `struct dentry *f_dentry`
>
> Utilisé pour accéder à l'inode: `filp->f_dentry->d_inode`.

Résumé pilotes caractère

- Définir vos opérations sur fichier (`fops`)
- Définir votre fonction d'init. du module et appeler `register_chrdev()`:
  - Donner un numéro majeur, ou 0 (automatique)
  - Donner vos fops
- Définir la fonction de sortie du module, et y appeler la fonction `unregister_chrdev()`
- Charger votre module
- Trouver le numéro majeur (si nécessaire) et créer l'entrée dans `/dev/`
- Utiliser le pilote !!

Gestion de la mémoire

kmalloc et kfree

- Allocateurs basiques, équivalents noyau des malloc et free de la glibc.
- `static inline void *kmalloc(size_t size, int flags);`

  `size`: quantité d'octets à allouer

  `flags`: priorité (voir la page suivante)
- `void kfree (const void *objp);`
- Exemple:

```
data = kmalloc(sizeof(*data), GFP_KERNEL);
```

Propriétés de kmalloc

- Rapide (à moins qu'il ne soit bloqué en attente de pages)
- N'initialise pas la zone allouée
- La zone allouée est contiguë en RAM physique
- Allocation par taille de 2^n-k (k: quelques octets de gestion)
  Ne demandez pas 1024 quand vous avez besoin de 1000 ! Vous recevriez 2048 !

Options pour kmalloc (1)

Définis dans `include/linux/gfp.h` (GFP: get_free_pages)

- `GFP_KERNEL`
  Allocation mémoire standard du noyau. Peut être bloquante. Bien pour la plupart des cas.
- `GFP_ATOMIC`
  Allocation de RAM depuis les gestionnaires d'interruption ou le code non liés aux processus utilisateurs. Jamais bloquante.
- `GFP_USER`
  Alloue de la mémoire pour les processus utilisateur. Peut être bloquante. Priorité la plus basse.
- GFP_NOIO
  Peut être bloquante, mais aucune action sur les E/S ne sera exécutée.
- GFP_NOFS
  Peut être bloquante, mais aucune opération sur les systèmes de fichier ne sera lancée.
- GFS_HIGHUSER
  Allocation de pages en mémoire haute en espace utilisateur. Peut être bloquante. Priorité basse.

Flags pour kmalloc (2)

Options supplémentaires (pouvant être ajoutés avec l'opérateur |)

- `__GFP_DMA`
  Allocation dans la zone DMA
- `__GFP_HIGHMEM`
  Allocation en mémoire étendue (x86 et sparc)
- `__GFP_REPEAT`
  Demande d'essayer plusieurs fois. Peut se bloquer, mais moins probable.
- `__GFP_NOFAIL`
  Ne doit pas échouer. N'abandonne jamais. Attention: à utiliser qu'en cas de nécessité!
- `__GFP_NORETRY`
  Si l'allocation échoue, n'essaie pas d'obtenir de page libre.

Allocation par pages

Plus appropriée que `kmalloc` pour les grosses tranches de mémoire:

- `unsigned long get_zeroed_page(int flags);`
  Retourne un pointeur vers une page libre et la remplit avec des zéros
- `unsigned long __get_free_page(int flags);`
  Identique, mais le contenu n'est pas initialisé
- `unsigned long __get_free_pages(int flags, unsigned long order);`
  Retourne un pointeur sur une zone mémoire de plusieurs pages continues en mémoire physique. `order`: $\log_2$(nombre_de_pages).

Libérer des pages

```
> void free_page(unsigned long addr);
> void free_pages(unsigned long addr,
                  unsigned long order);
```

Utiliser le même ordre que lors de l'allocation.

Mapper des adresses physiques

`vmalloc` et `ioremap` peuvent être utilisés pour obtenir des zones mémoire continues dans l'espace d'adresse *virtuel* (même si les pages peuvent ne pas être continues en mémoire physique).

> `void *vmalloc(unsigned long size);`

> `void vfree(void *addr);`

> `void *ioremap(unsigned long phys_addr, unsigned long size);`

  Ne fait pas d'allocation. Fait correspondre le segment donné en mémoire physique dans l'espace d'adressage virtuel.

> `void iounmap(void *address);`

Utilitaires pour la mémoire

- `void * memset(void * s, int valeur, size_t taille);`

  Remplit une région mémoire avec la valeur donnée.

- ```
  void * memcpy(void * dest,
                const void *src,
                size_t count);
  ```

  Copie une zone mémoire vers une autre.

  Utiliser `memmove` avec des zones qui se chevauchent.

- De nombreuses fonctions équivalentes à celles de la glibc sont définies dans `include/linux/string.h`


  ```

Mémoire et ports d'E/S

Demander des ports d'E/S

/proc/ioports example

```
0000-001f : dma1
0020-0021 : pic1
0040-0043 : timer0
0050-0053 : timer1
0060-006f : keyboard
0070-0077 : rtc
0080-008f : dma page reg
00a0-00a1 : pic2
00c0-00df : dma2
00f0-00ff : fpu
0100-013f : pcmcia_socket0
0170-0177 : ide1
01f0-01f7 : ide0
0376-0376 : ide1
0378-037a : parport0
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial
0800-087f : 0000:00:1f.0
  0800-0803 : PM1a_EVT_BLK
  0804-0805 : PM1a_CNT_BLK
  0808-080b : PM_TMR
  0820-0820 : PM2_CNT_BLK
  0828-082f : GPE0_BLK
...
```

> `struct resource *request_region(`
>
> `unsigned long start,`
>
> `unsigned long len,`
>
> `char *name);`

Essaie de réserver la région donnée et retourne NULL en cas d'échec. Exemple:

```
request_region(0x0170, 8, "ide1");
```

> `void release_region(`
>
> `unsigned long start,`
>
> `unsigned long len);`

> Regarder `include/linux/ioport.h` et `kernel/resource.c`

Lire / écrire sur les ports d'E/S

L'implémentation des fonctions suivantes et le type *unsigned* peuvent varier suivant la plate-forme !

octets

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

mots

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

"long" integers

```
unsigned inl(unsigned port);
void outl(unsigned long word, unsigned port);
```

Lire / écrire une chaîne sur les ports d'E/S

Plus efficace que la boucle C correspondante, si le processeur supporte de telles opérations!

byte strings

```
void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
```

word strings

```
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
```

long strings

```
void inbsl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
```

Demander de la mémoire d'E/S

/proc/iomem example

```
00000000-0009efff : System RAM
0009f000-0009ffff : reserved
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000f0000-000fffff : System ROM
00100000-3ffadfff : System RAM
  00100000-0030afff : Kernel code
  0030b000-003b4bff : Kernel data
3ffae000-3fffffff : reserved
40000000-400003ff : 0000:00:1f.1
40001000-40001fff : 0000:02:01.0
  40001000-40001fff : yenta_socket
40002000-40002fff : 0000:02:01.1
  40002000-40002fff : yenta_socket
40400000-407fffff : PCI CardBus #03
40800000-40bfffff : PCI CardBus #03
40c00000-40ffffff : PCI CardBus #07
41000000-413fffff : PCI CardBus #07
a0000000-a0000fff : pcmcia_socket0
a0001000-a0001fff : pcmcia_socket1
e0000000-e7ffffff : 0000:00:00.0
e8000000-efffffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
...
```

- Fonctions équivalentes avec la même interface
-
```
struct resource *request_mem
region(
    unsigned long start,
    unsigned long len,
    char *name);
```
-
```
void release_mem_region(
    unsigned long start,
    unsigned long len);
```

Choisir un intervalle d'E/S

- Les limites de la mémoire et des ports d'E/S peuvent être passés comme paramètres de module. Un moyen facile de définir ces paramètre est au travers de `/etc/modprobe.conf`
- Les modules peuvent aussi essayer de trouver des zones libres par eux-mêmes (en faisant plusieurs appels à `request_region`).

Différences avec la mémoire standard

- Écriture et lecture sur la mémoire peuvent être mis en cache.
- Le compilateur peut choisir d'écrire la valeur dans un registre du processeur, et ne jamais l'écrire dans la mémoire principale.
- Le compilateur peut décider d'optimiser ou réordonner les instructions de lecture / écriture.

Eviter les problèmes d'accès aux E/S

- Le cache sur la mémoire et les ports d'E/S est désactivé, soit par le hardware ou par le code d'init Linux.
- Linux fournit les Barrières Mémoire pour empêcher le compilateur de réordonnancer les accès:

Dépendant de l'architecture

```
#include <asm/system.h>
void rmb(void);
void wmb(void);
void mb(void);
```

Indépendant

```
#include <asm/kernel.h>
void barrier(void);
```

Mémoire mappée directement

- Dans certaines architectures (principalement MIPS), la mémoire d'E/S peut être directement mappée dans l'espace d'adressage physique.
- Dans ce cas, les pointeurs d'E/S ne doivent pas être déréférencés.
- Pour éviter les problèmes de portabilité à travers les architectures, les fonctions suivantes peuvent être utilisées:

```
unsigned read[b|w|l](address);
void writeb[b|w|l](unsigned value, address);

void memset_io(address, value, count);
void memcpy_fromio(dest, source, num);
void memcpy_toio(dest, source, num);
```

Mapper la mémoire d'E/S en mémoire virtuelle

- Pour accéder à la mémoire d'E/S, les pilotes ont besoin d'une adresse virtuelle que le processeur peut gérer.
- Les fonctions `ioremap` permettent cela:

```
#include <asm/io.h>

void *ioremap(unsigned long phys_addr,
              unsigned long size);
void *ioremap_nocache(unsigned long phys_addr,
                      unsigned long size);
void iounmap(void *address);
```

- Attention: vérifiez que `ioremap` ne retourne pas NULL !

Drivers Linux

S'endormir

Comment s'endormir

L'endormissement est nécessaire lorsqu'un processus utilisateur attend des données qui ne sont pas encore prêtes. Il est alors placé dans une queue/file d'attente.

Déclarer la queue

> `DECLARE_WAIT_QUEUE_HEAD (module_queue);`

Plusieurs moyens d'endormir un processus

> `sleep_on()`
> Ne peut pas être interrompu !

> `interruptible_sleep_on()`
> Peut être interrompu par un signal

> `sleep_on_timeout()`
> `interruptible_sleep_on_timeout()`
> Similaire à ci-dessus, mais avec un délai d'expiration.

> `wait_event()`
> `wait_event_interruptible()`
> Dort jusqu'à ce qu'une condition soit vérifiée.

Utilisez seulement les commandes interruptibles ! Les autres sont rarement nécessaires.

Se réveiller !

- `wake_up(&queue);`
  Réveille tous les processus attendant dans la queue donnée
- `wake_up_interruptible(&queue);`
  Réveille seulement les processus interruptibles
- `wake_up_sync(&queue);`
  Ne réordonnance pas lorsque vous savez qu'un autre processus est sur le point de s'endormir, car dans ce cas un réordonnancement va de toute façon se produire.

Gestion des interruptions

Les interruptions, pour quoi faire ?

- Les interruptions internes au processeur sont par exemple utilisées pour l'ordonnancement, nécessaire au multi-tâche.
- Les interruptions externes sont utiles car la plupart des périphériques internes ou externes sont plus lents que le processeur. Dans ce cas, mieux vaut ne pas laisser le processeur en attente active sur des données. Lorsque le périphérique est à nouveau près, il envoie une interruption pour demander l'attention du processeur.

Enregistrer un gestionnaire d'interruption (1)

Défini dans `include/linux/interrupt.h`

```
> int request_irq(
        unsigned int irq, /* canal irq demandé */
        irqreturn_t (*handler) (int, void *,
            struct pt_regs *), /* gestionnaire d'inter. */
        unsigned long irq_flags, /* masque d'options */
        const char* devname, /* nom enregistré */
        void* dev_id) /* utilisé lorsque l'irq est partagée */

> void free_irq(
        unsigned int irq,
        void *dev_id);
```

Enregistrer un gestionnaire d'interruption (2)

Bits pouvant être définis dans `irq_flags` (combinables)

- `SA_INTERRUPT`
  Gestionnaire d'interruption "rapide". Fonctionne avec les interruptions désactivées. Utilité limitée à des cas spécifiques (tels que les interruptions timer).
- `SA_SHIRQ`
  Le canal d'interruption peut être partagé par plusieurs périphériques.
- `SA_SAMPLE_RANDOM`
  Les interruptions peuvent être utilisées pour contribuer à l'entropie du système (`/dev/random` et `/dev/urandom`), afin de générer de bons nombres aléatoires. Ne l'utilisez pas si votre périphérique a un comportement prédictif !

Quand enregistrer le gestionnaire

- Soit à l'initialisation du pilote:
  consomme beaucoup de canaux IRQ !
- Ou bien à l'ouverture du fichier de périphériques:
  permet de sauver des canaux IRQ libres.
  Besoin de compter le nombre de fois où le périphérique est ouvert, pour être capable de libérer les canaux IRQ lorsque le périphérique n'est plus utilisé.

Information sur les gestionnaires installés

/proc/interrupts

```
           CPU0
  0:    5616905          XT-PIC  timer  # Nom enregistré
  1:       9828          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  3:    1014243          XT-PIC  orinoco_cs
  7:        184          XT-PIC  Intel 82801DB-ICH4
  8:          1          XT-PIC  rtc
  9:          2          XT-PIC  acpi
 11:     566583          XT-PIC  ehci_hcd, uhci_hcd,...
 12:       5466          XT-PIC  i8042
 14:     121043          XT-PIC  ide0
 15:     200888          XT-PIC  ide1
NMI:          0          # Interruptions non masquables
ERR:          0
```

Nombre total d'interruptions

```
cat /proc/stat | grep intr
intr 8190767 6092967 10377 0 1102775 5 2 0 196
  ...
```

Nombre total d'interruptions

Total IRQ1

IRQ2

IRQ3 ...

Détection du canal d'interruption (1)

- Certains périphériques annoncent leur canal IRQ dans un registre
- Certains périphériques ont toujours le même comportement: vous pouvez déduire leur canal IRQ
- Détection manuelle:
  - Enregistrez votre gestionnaire d'inter. pour tous les canaux possibles
  - Demander une interruption
  - Dans le gestionnaire appelé, enregistrer le numéro d'IRQ dans une variable globale
  - Réessayez si aucune interruption n'a été reçue
  - Désenregistrez les gestionnaires non utilisés

Détection du canal d'interruption (2)

Outils de détection du noyau:

- `mask = probe_irq_on();`
- Activez les interruptions sur le périphérique
- Désactivez les interruptions sur le périphérique
- `irq = probe_irq_off(mask);`
  - > 0: numéro d'IRQ unique trouvé
  - = 0: pas d'interruption. Essayez à nouveau !
  - < 0: plusieurs interruptions reçues. Essayez à nouveau !

Le travail du gestionnaire d'interruption

- Acquitter l'interruption au périphérique (sinon plus aucune autre interruption ne sera générée)
- Lire/écrire des donnée du/sur le périphérique
- Réveiller tout processus attendant la fin de l'opération de lecture/écriture:

```
wake_up_interruptible(&module_queue);
```

Contraintes du gestionnaire d'interruption

- Ne s'exécute pas dans le contexte utilisateur:
  Ne peut pas transférer des donnés de ou vers l'espace utilisateur
- Ne peut pas exécuter d'actions pouvant s'endormir:
  Besoin d'allouer de la mémoire avec `GFP_ATOMIC`
- Ne peut pas appeler `schedule()`
- Doit terminer son travail suffisamment rapidement:
  il ne peut pas bloquer les interruptions trop longtemps.

Prototype d'un gestionnaire

```
irqreturn_t (*handler) (
  int,                          /* Numéro d'irq */
  void *dev_id,                 /* Pointeur utilisé pour garder la trace
                                du device correspondant. Utile quand
                                plusieurs périphériques sont gérés par le
                                même module */
  struct pt_regs *regs          /* snapshot des registres du cpu, rarement
                                nécessaire */
);
```

Valeur retournée:

- `IRQ_HANDLED`: interruption reconnue et gérée
- `IRQ_NONE`: pas sur un périphérique géré par le module. Permet de partager des canaux d'interruption et/ou de reporter de fausses interruptions au noyau.

Traitement parties haute et basse (1)

- *Partie haute*: le gestionnaire doit se terminer le plus vite que possible. Une fois qu'il a acquitté l'interruption, il planifie le reste du travail gérant les données et prenant du temps, pour une exécution future.
- *Partie basse*: termine le reste du travail du gestionnaire. Traite les données, et ensuite réveille tout les processus utilisateur en attente.
  Implémenté au mieux par les *tasklets*.

Traitement parties haute et basse (2)

- Déclarer la tasklet dans le fichier source du module:

```
DECLARE_TASKLET (module_tasklet,      /* nom */
                 module_do_tasklet,   /* fonction */
                 0                    /* données */
);
```

- Planifier la tasklet dans la partie haute du gestionnaire:

```
tasklet_schedule(&module_do_tasklet);
```

Résumé de la gestion des interruptions

- Trouver un numéro d'interruption (si possible)
- Activer les interruptions sur le périphérique
- Détecter le numéro d'interruption utilisé par le périphérique, en scrutant les différents possibilités, si nécessaire.
- Enregistrer le gestionnaire d'interruption avec le numéro d'IRQ identifié.
- Une fois que le gestionnaire d'interruption est appelé, acquitter l'interruption.
- Dans le gestionnaire, planifier la tasklet gérant les données
- Dans la tasklet, gérer les données
- Dans la tasklet, réveiller les processus utilisateur en attente
- Désenregistrer le gestionnaire si le périphérique est fermé.

Drivers Linux

mmap

mmap

- Répond aux requêtes de la fonction `mmap` de la glibc:

```
void * mmap(void *start, size_t length, int prot,
            int flags, int fd, off_t offset);
int munmap(void *start, size_t length);
```

- Permet aux programmes utilisateurs d'accéder directement à la mémoire du périphérique.
- Utilisé par des programmes comme le serveur X-Window. Plus rapide que les autres méthodes (comme écrire dans le fichier `/dev` correspondant) pour les applications utilisateur à fort besoin en bande passante.

Zones de Mémoire Virtuelle (1)

Zone de Mémoire Virtuelle (Virtual Memory Areas): zone contiguë dans la mémoire virtuelle d'un processus, avec les mêmes permissions.

```
> cat /proc/1/maps  (processus init)
```

```
Début          fin            perm  décalage  majeur:mineur inode     Nom du fichier mappé
00771000-0077f000 r-xp 00000000 03:05 1165839    /lib/libselinux.so.1
0077f000-00781000 rw-p 0000d000 03:05 1165839    /lib/libselinux.so.1
0097d000-00992000 r-xp 00000000 03:05 1158767    /lib/ld-2.3.3.so
00992000-00993000 r--p 00014000 03:05 1158767    /lib/ld-2.3.3.so
00993000-00994000 rw-p 00015000 03:05 1158767    /lib/ld-2.3.3.so
00996000-00aac000 r-xp 00000000 03:05 1158770    /lib/tls/libc-2.3.3.so
00aac000-00aad000 r--p 00116000 03:05 1158770    /lib/tls/libc-2.3.3.so
00aad000-00ab0000 rw-p 00117000 03:05 1158770    /lib/tls/libc-2.3.3.so
00ab0000-00ab2000 rw-p 00ab0000 00:00 0
08048000-08050000 r-xp 00000000 03:05 571452     /sbin/init  programme
08050000-08051000 rw-p 00008000 03:05 571452     /sbin/init  données, pile
08b43000-08b64000 rw-p 08b43000 00:00 0
f6fdf000-f6fe0000 rw-p f6fdf000 00:00 0
fefd4000-ff000000 rw-p fefd4000 00:00 0
ffffe000-fffff000 ---p 00000000 00:00 0
```

Zones de Mémoire Virtuelle (2)

Exemple du serveur X (extrait)

```
Début           fin             perm  décalage  majeur:mineur  inode    Nom du fichier mappé
08047000-081be000 r-xp 00000000 03:05 310295     /usr/X11R6/bin/Xorg
081be000-081f0000 rw-p 00176000 03:05 310295     /usr/X11R6/bin/Xorg
...
f4e08000-f4f09000 rw-s e0000000 03:05 655295     /dev/dri/card0
f4f09000-f4f0b000 rw-s 4281a000 03:05 655295     /dev/dri/card0
f4f0b000-f6f0b000 rw-s e8000000 03:05 652822     /dev/mem
f6f0b000-f6f8b000 rw-s fcff0000 03:05 652822     /dev/mem
```

mmap simple

Pour autoriser les opérations `mmap()`, le pilote a juste besoin de créer des pages de mémoire mappant une zone physique.

Cela peut être fait avec la fonction suivante (`linux/mm.h`) à appeler dans une fonction *`driver`*`_mmap`:

```
int remap_page_range(
   struct vm_area_struct *vma,
   unsigned long from, /* Virtual */
   unsigned long to, /* Physical */
   unsigned long size, pgprot_t prot);
```

Cette fonction est alors à ajouter à la structure file_operations du pilote.

Exemple: `drivers/char/mem.c`

DMA (Accès Direct à la Mémoire)

Utilisation de la DMA

Synchrone

- Un processus utilisateur appelle la méthode de lecture d'un pilote. Celui-ci alloue un tampon DMA et demande au matériel de copier ces données. Le processus est placé en veille.
- Le matériel copie les données et provoque une interruption à la fin de la copie.
- Le gestionnaire récupère les données du tampon et réveille le processus en attente.

Asynchrone

- Le matériel envoie une interruption pour annoncer de nouvelles données.
- Le gestionnaire alloue un tampon DMA et dit au matériel où transférer les données.
- Le matériel écris les données et lève une nouvelle interruption.
- Le gestionnaire récupère les nouvelles données et réveille les processus nécessaires.

Contraintes mémoire

- Besoin d'utiliser de la mémoire contiguë dans l'espace physique
- Peut utiliser n'importe quelle mémoire allouée par `kmalloc` ou `__get_free_pages`
- E/S bloc ou réseau: possibilité d'utiliser des tampons spécialisés, conçus pour être compatibles avec la DMA.
- Ne peut pas utiliser la mémoire `vmalloc`
  (il faudrait configurer la DMA pour chaque page)
- Utiliser la DMA ne supprime pas le besoin de barrières mémoire. Autrement, votre compilateur peut mélanger l'ordre des lectures et des écritures.

Mappages DMA permanents ou de flux

- Mappages permanents («consistants»)
  Alloués pour toute la durée de chargement du module.
- Mappages de flux
  Configurés à chaque transfert.
  Cela permet de garder libres des registres matériels pour la DMA.
  Certaines optimisation sont aussi disponibles.

API générique DMA

- La plupart des sous-systèmes fournissent leur propre API DMA. Suffisant pour la plupart des besoins.
- Documentation/DMA-API.txt
  Description de l'API DMA générique Linux, en parallèle avec l'API PCI DMA correspondante.

  Un document utile à montrer lors de ce cours !

Nouveau modèle de périphériques

Caractéristiques (1)

- A été créé au départ pour rendre la gestion de l'alimentation plus simple. Va maintenant bien au delà
- Utilisé pour représenter l'architecture et l'état du système
- A une représentation dans l'espace utilisateur: sysfs
  Désormais c'est l'interface préférée avec l'espace utilisateur (au lieu de /proc)
- Facile à implémenter grâce à l'interface device: `include/linux/device.h`

Caractéristiques (2)

Permet de voir le système depuis différents points de vue:

- Depuis les périphériques existant dans le système: leur état d'alimentation, le bus auquel ils sont attachés, et le pilote qui en est responsable.
- Depuis la structure du bus système: quel bus est connecté à quel bus (par exemple contrôleur de bus USB sur le bus PCI), les périphériques existant et ceux potentiellement acceptés (avec leur pilotes)
- Depuis les pilotes disponibles: quels périphériques et quel types de bus sont supportés.
- Depuis différentes "classes" de périphériques: "input", "net", "sound"... Périphériques existant pour chaque classe. Permet de trouver tous les périphériques d'entrée sans savoir où ils sont connectés physiquement.

sysfs

- Représentation dans l'espace utilisateur du «Modèle de périphérique».
- Configurer par
  `CONFIG_SYSFS=y` (Filesystems -> Pseudo filesystems)
- Monter par
  `mount -t sysfs /sys /sys`
- Passez du temps à explorer /sys sur votre station de travail !

Outils sysfs

http://linux-diag.sourceforge.net/Sysfsutils.html

- `libsysfs` – Le but de cette librairie est de fournir une interface stable et pratique pour obtenir des informations sur les périphériques système exportés à travers sysfs. Utilisé par `udev` (voir plus loin)
- `systool` – Un utilitaire bâti au dessus de libsysfs qui liste les périphériques par bus, classe et topologie.

La structure «device»

Déclaration

- La structure de donnée de base est `struct device`, définie dans `include/linux/device.h`
- En pratique, vous utiliserez plutôt une structure correspondant au bus auquel votre périphérique est attaché: `struct pci_dev`, `struct usb_device`...

Enregistrement

- Dépend toujours du type de périphérique, des fonctions spécifiques (de)d'(dés)enregistrement sont fournies

Attributs de périphériques

Les attributs du périphériques peuvent être lus/écrits depuis l'espace utilisateur

```
struct device_attribute {
   struct attribute        attr;
   ssize_t (*show)(struct device * dev, char * buf, size_t count, loff_t off);
   ssize_t (*store)(struct device * dev, const char * buf, size_t count, loff_t off);
};

#define DEVICE_ATTR(name,mode,show,store)
```

Ajouter / enlever un fichier

```
int device_create_file(struct device *device, struct device_attribute * entry);
void device_remove_file(struct device * dev, struct device_attribute * attr);
```

Exemple

```
/* Créé un fichier nommé "power" avec un mode 0644 (-rw-r--r--) */

DEVICE_ATTR(power,0644,show_power,store_power);
device_create_file(dev,&dev_attr_power);
device_remove_file(dev,&dev_attr_power);
```

La structure «device_driver»

Déclaration

```
struct device_driver {
    /* Omitted a few internals */
    char          *name;
    struct bus_type         *bus;
    int     (*probe)        (struct device * dev);
    int     (*remove)       (struct device * dev);
    void    (*shutdown)      (struct device * dev);
    int     (*suspend)       (struct device * dev, u32 state, u32 level);
    int     (*resume)       (struct device * dev, u32 level);
};
```

Enregistrement

```
extern int driver_register(struct device_driver * drv);
extern void driver_unregister(struct device_driver * drv);
```

Attributs

Disponibles de la même manière

Références pour le «Device Model»

La documentation dans les sources du noyau est très utile et très claire !

- `Documentation/driver-model/`

```
binding.txt  class.txt     driver.txt overview.txt
porting.txt   bus.txt   device.txt   interface.txt
platform.txt
```

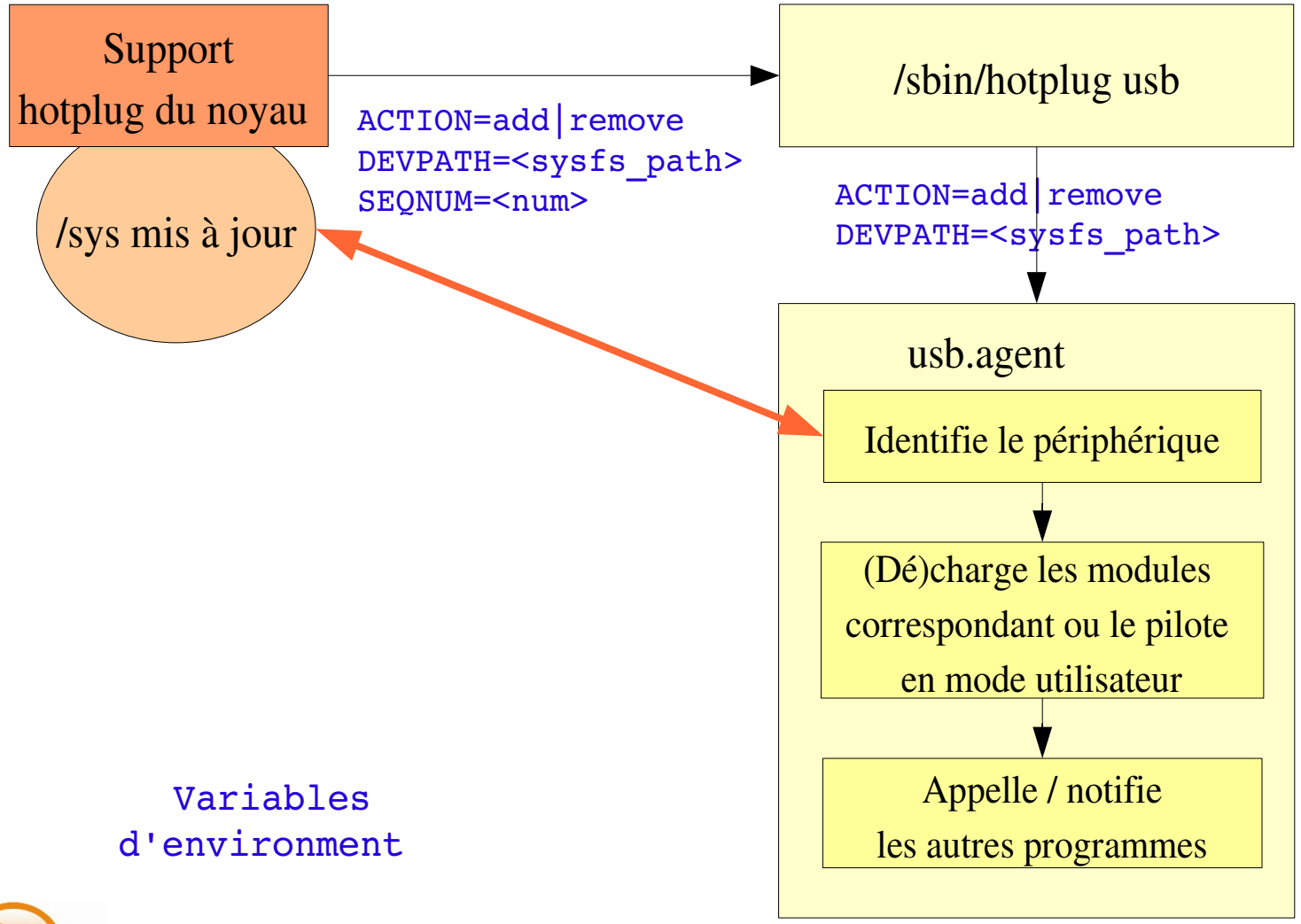
- `Documentation/filesystems/sysfs.txt`

Branchement à chaud (hotplug)

Aperçu du hotplug

- Introduit dans le noyau Linux 2.4. USB a été le pionnier.
- Mécanismes noyau pour notifier les programmes de l'espace utilisateur qu'un périphérique a été inséré ou enlevé.
- Des scripts dans l'espace utilisateur prennent ensuite soin d'identifier le matériel et d'insérer/enlever les modules requis.
- Linux 2.6: l'identification des périphériques est simplifiée grâce à sysfs.
- Rend possible le chargement de micrologiciel (firmware).
- Permet d'avoir des pilotes en mode utilisateur (par exemple libsane).
- Configuration dans le noyau:
  `CONFIG_HOTPLUG=y` (section "General setup")

Exemple de flux hotplug



Fichiers pour hotplug

`/lib/modules/*/modules.*map`
sortie de depmod

`/proc/sys/kernel/hotplug`
specifie le chemin du programme hotplug

`/sbin/hotplug`
programme hotplug (par défaut)

`/etc/hotplug/*`
fichiers hotplug

`/etc/hotplug/NAME*`
fichiers spécifiques aux sous-systèmes, pour les agents

`/etc/hotplug/NAME/DRIVER`
scripts de configuration des pilotes, invoqués par les agents

`/etc/hotplug/usb/DRIVER.usermap`
données pour depmod pour les pilotes en mode utilisateur

`/etc/hotplug/NAME.agent`
Agents spécifiques à des sous-systèmes de hotplug.

Références sur hotplug

- Page du projet et documentation
  http://linux-hotplug.sourceforge.net/
- Liste de diffusion:
  http://lists.sourceforge.net/lists/listinfo/linux-hotplug-deve

udev

Gestion des fichiers de périphériques dans l'espace utilisateur

Problèmes et limitations de /dev

- Sur Red Hat 9, il y avait 18000 entrées dans `/dev`!
  A l'installation du système, toutes les entrées possibles des périphériques doivent être créées.
- Besoin d'une autorité pour assigner les major numbers
  http://lanana.org/: Linux Assigned Names and Numbers Authority
- Pas assez de nombres dans 2.4, la limite a été étendue dans 2.6
- L'espace utilisateur ne sait pas quels périphériques sont présents dans le système.
- L'espace utilisateur ne sait pas associer une entrée dans `/dev` avec le périphérique auquel elle se rapporte

La solution devfs et ses limitations

- Montre seulement les périphériques présents
- Mais utilise des noms différents à ceux de `/dev`, ce qui pose des problèmes dans les scripts.
- Mais aucune flexibilité dans le nom des devices (par rapport à `/dev/`) i.e. le 1er disque IDE s'appelle soit `/dev/hda`, soit `/dev/ide/hd/c0b0t0u0`.
- Mais ne permet pas l'allocation dynamique des nombres majeur et mineur.
- Mais requiert de stocker la politique de nommage des périphériques dans la mémoire du noyau. Ne peut pas être swappée!

Caractéristiques de udev

Tire partie à la fois de hotplug et de sysfs

- Entièrement dans l'espace utilisateur
- Créé automatiquement les entrées pour les périphériques (par défaut dans `/udev`)
- Appelé par `/sbin/hotplug`, utilise les informations de sysfs (notamment les nombres Majeur/Mineur)
- Ne requiert aucun changement dans le code du pilote
- Petite taille

La boîte à outils de udev (1)

Composants principaux:

- `udevsend` (8KB dans FC 3)

  Gère les événements de `/sbin/hotplug`, et les envoie à `udevd`

- `udevd` (12KB dans FC 3)

  Réordonne les événements de hotplug, avant d'appeler les instance de `udev` pour chacun d'eux.

- `udev` (68KB dans FC 3)

  Créé ou efface les fichiers périphériques et ensuite exécute les programmes dans `/etc/dev.d/`

La boîte à outils de udev (2)

Autres utilitaires

- `udevinfo` (48KB dans FC 3)
  Permet aux utilisateurs d'interroger la base de données de udev
- `udevstart` (fonctionnalité apportée par `udev`)
  Remplit le répertoire initial des périphériques avec des entrées valides trouvées dans l'arbre de périphériques de sysfs.
- `udevtest <chemin_dev_sysfs>` (64KB dans FC 3)
  Simule une exécution de `udev` pour tester les règles configurées

Le fichier de configuration de udev

`/etc/udev/udev.conf`

Facile à éditer et à configurer

- Répertoire des fichiers de périphériques (`/udev`)
- Base de données udev (`/dev/.udev.tdb`)
- Règles udev (`/etc/udev/rules.d/`)
  Permissions udev (`/etc/udev/permissions.d/`)
- mode par défaut (`0600`), utilisateur par défaut (`root`) et groupe (`root`), si pas trouvés dans les permissions de udev.
- Activation des traces (log) (`yes`)
  Les messages de déboguage sont dans `/var/log/messages`

Capacités de nommage de udev

Le nom des fichiers de périphériques peut être défini

- depuis un label ou un numéro de série
- depuis un numéro de périphérique sur le bus
- depuis une position dans la topologie du bus
- depuis un nom du noyau

udev peut aussi créé des liens symboliques vers d'autres fichiers périphériques

Exemple de fichier de règles pour udev

```
# Si /sbin/scsi_id retourne "OEM 0815", le périphérique sera appelé disk1
BUS="scsi", PROGRAM="/sbin/scsi_id", RESULT="OEM 0815", NAME="disk1"

# Imprimante USB appelée lp_color
BUS="usb", SYSFS{serial}="W09090207101241330", NAME="lp_color"

# Un disque SCSI avec un numéro de vendeur/modèle spécifique devient boot
BUS="scsi", SYSFS{vendor}="IBM", SYSFS{model}="ST336", NAME="boot%n"

# La carte son avec l'id 00:0b.0 sur le bus PCI sera appelée dsp
BUS="pci", ID="00:0b.0", NAME="dsp"

# La souris USB sur le 3ème port du 2nd hub sera appelée mouse1
BUS="usb", PLACE="2.3", NAME="mouse1"

# ttyUSB1 doit tjrs être appelé pda et avoir 2 autres liens symboliques
KERNEL="ttyUSB1", NAME="pda", SYMLINK="palmtop handheld"

# Webcams USB multiples avec des liens appelés webcam0, webcam1, ...
BUS="usb", SYSFS{model}="XV3", NAME="video%n", SYMLINK="webcam%n"
```

Exemple de fichier de permissions udev

```
#name:user:group:mode
input/*:root:root:644
ttyUSB1:0:8:0660
video*:root:video:0660
dsp1:::0666
```

/etc/dev.d/

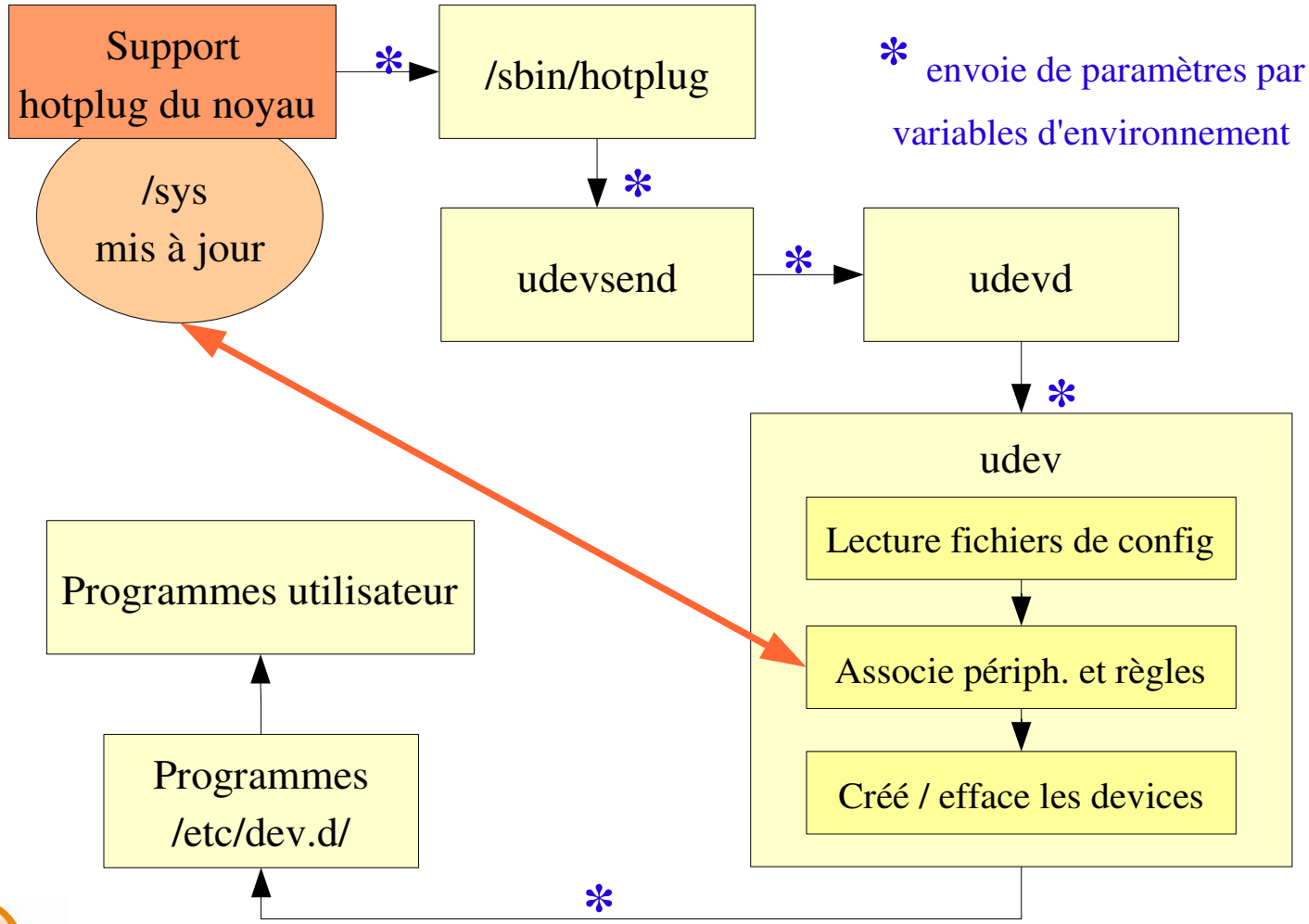
Après la création, destruction, renommage des noeuds de périphériques, udev peut appeler des programmes dans l'ordre suivant de priorité:

- `/etc/dev.d/$(DEVNAME)/*.dev`
- `/etc/dev.d/$(SUBSYSTEM)/*.dev`
- `/etc/dev.d/default/*.dev`

Les programmes de chaque répertoire sont triés par ordre alphabétique.

Cela permet de notifier les applications utilisateurs de changements au niveau des périphériques.

Résumé de udev



Liens udev

- Sources
  http://kernel.org/pub/linux/utils/kernel/hotplug/
- Liste de diffusion:
  linux-hotplug-devel@lists.sourceforge.net
- Présentation de udev de Greg Kroah-Hartman
  http://www.kroah.com/linux/talks/oscon_2004_udev/
- Documentation udev de Greg Kroah-Hartman
  http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-C

Linux USB Drivers

Purpose of this course

Learn how to implement Linux drivers for some of the most complex USB devices!

Buy yours on http://www.thinkgeek.com/stuff/41/fundue.shtml!

Course prerequisites

- Fondue cheese 
- Good knowledge about Linux device driver development. Most notions which are not USB specific are covered in our http://free-electrons.com/training/drivers course.
- To create real, working drivers: a good knowledge about the USB devices you want to write drivers for. A good knowledge about USB specifications too.

Linux USB basics

Linux USB drivers

USB drivers (1)

USB core drivers

> Architecture independent kernel subsystem.
> Implements the USB bus specification.
> Outside the scope of this training.

USB host drivers

> Different drivers for each USB control hardware.
> Usually available in the Board Support Package.
> Architecture and platform dependent.
> Not covered yet by this training.

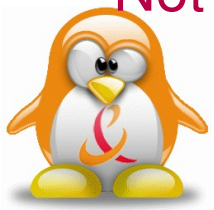
USB drivers (2)

USB device drivers

- Drivers for devices on the USB bus.
  The main focus of this course!
- Platform independent: when you use Linux on an embedded platform, you can use any USB device supported by Linux (cameras, keyboards, video capture, wi-fi dongles...).

USB device controller drivers

- For Linux systems with just a USB device controller (frequent in embedded systems).
  Not covered yet by this course.



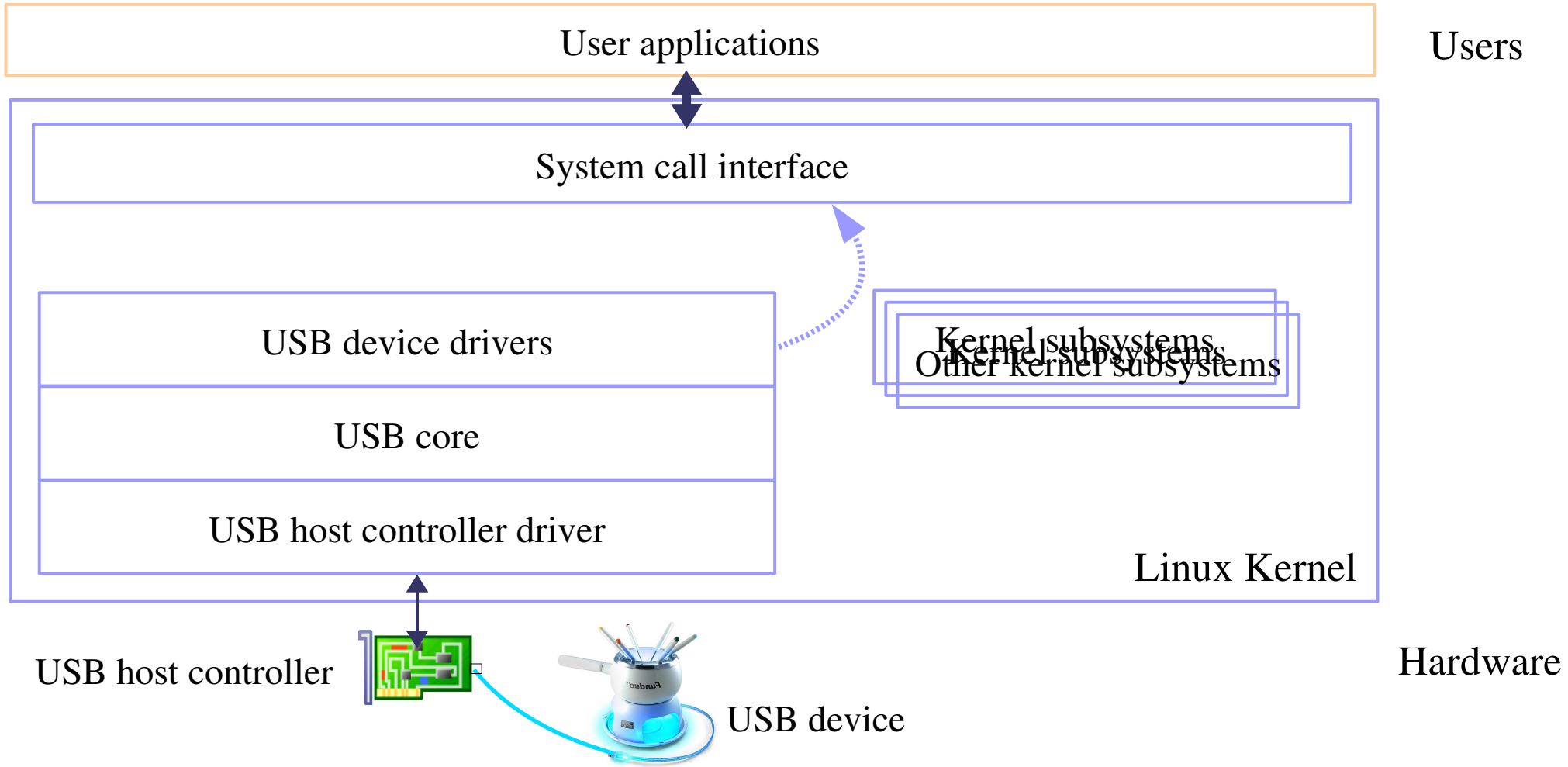
USB gadget drivers

Drivers for Linux systems with a USB device controller

- Typical example: digital cameras.
  You connect the device to a PC and see the camera
  as a USB storage device.
- USB device controller driver:
  Platform dependent. Supports the chip connecting to the USB bus.
- USB gadget drivers, platform independent. Examples:
  Ethernet gadget: implements networking through USB
  Storage gadget: makes the host see a USB storage device
  Serial gadget: for terminal-type of communication.

See Documentation/DocBook/gadget/ in kernel sources.

Linux USB support overview



USB host controllers - OHCI and UHCI

2 competing Host Control Device (HCD) interfaces

- OHCI - Open Host Controller Interface
  Compaq's implementation adopted as a standard for USB 1.0 and 1.1 by the USB Implementers Forum (USB-IF).
  Also used for Firewire devices.
- UHCI - Universal Host Controller Interface.
  Created by Intel, insisting that other implementers use it and pay royalties for it. Only VIA licensed UHCI, and others stuck to OHCI.

This competition required to test devices for both host controller standards!

For USB 2.0, the USB-IF insisted on having only one standard.

USB host controllers - EHCI

EHCI - Extended Host Controller Interface.

- For USB 2.0. The only one to support high-speed transfers.
- Each EHCI controller contains four virtual HCD implementations to support Full Speed and Low Speed devices.
- On Intel and VIA chipsets, virtual HCDs are UHCI. Other chipset makers have OHCI virtual HCDs.

USB transfer speed

- Low-Speed: up to 1.5 Mbps
  Since USB 1.0
- Full-Speed: up to 12 Mbps
  Since USB 1.1
- Hi-Speed: up to 480 Mbps
  Since USB 2.0

Linux USB basics

USB devices

USB descriptors

Operating system independent. Described in the USB specification

- Device - Represent the devices connected to the USB bus.
  Example: USB speaker with volume control buttons.
- Configurations - Represent the state of the device.
  Examples: Active, Standby, Initialization
- Interfaces - Logical devices.
  Examples: speaker, volume control buttons.
- Endpoints - Unidirectional communication pipes.
  Either `IN` (device to computer) or `OUT` (computer to device).

Control endpoints

- Used to configure the device, get information about it, send commands to it, retrieve status information.
- Simple, small data transfers.
- Every device has a control endpoint (endpoint 0), used to configure the device at insertion time.
- The USB protocol guarantees that the corresponding data transfers will always have enough (reserved) bandwidth.

Interrupt endpoints

- Transfer small amounts of data at a fixed rate each time the hosts asks the device for data.
- Guaranteed, reserved bandwidth.
- For devices requiring guaranteed response time, such as USB mice and keyboards.
- Note: different than hardware interrupts. Require constant polling from the host.

Bulk endpoints

- Large sporadic data transfers
  using all remaining available bandwidth.
- No guarantee on bandwidth or latency.
- Guarantee that no data is lost.
- Typically used for printers, storage or network devices.

Isochronous endpoints

- Also for large amounts of data.
- Guaranteed speed
  (often but not necessarily as fast as possible).
- No guarantee that all data makes it through.
- Used by real-time data transfers (typically audio and video).

The `usb_endpoint_descriptor` structure (1)

The `usb_endpoint_descriptor` structure contains all the USB-specific data announced by the device itself.

Here are useful fields for driver writers:

> `__u8 bEndpointAddress`:

USB address of the endpoint.

It also includes the direction of the endpoint. You can use the `USB_ENDPOINT_DIR_MASK` bitmask to tell whether this is a `USB_DIR_IN` or `USB_DIR_OUT` endpoint. Example:

```
if ((endpoint->desc.bEndpointAddress &
USB_ENDPOINT_DIR_MASK) == USB_DIR_IN)
```

The `usb_endpoint_descriptor` structure (2)

- `__u8 bmAttributes`:
  The type of the endpoint. You can use the `USB_ENDPOINT_XFERTYPE_MASK` bitmask to tell whether the type is `USB_ENDPOINT_XFER_ISOC`, `USB_ENDPOINT_XFER_BULK`, `USB_ENDPOINT_XFER_INT` or `USB_ENDPOINT_XFER_CONTROL`.
- `__u8 wMaxPacketSize`:
  Maximum size in bytes that the endpoint can handle. Note that if greater sizes are used, data will be split in `wMaxPacketSize` chunks.
- `__u8 bInterval`:
  For interrupt endpoints, device polling interval (in milliseconds).

Note that the above names do not follow Linux coding standards.
The Linux USB implementation kept the original name from the USB specification (http://www.usb.org/developers/docs/).

Interfaces

- Each interface encapsulates a single high-level function (USB logical connection). Example (USB webcam): video stream, audio stream, keyboard (control buttons).
- One driver is needed for each interface!
- Alternate settings: each USB interface may have different parameter settings. Example: different bandwidth settings for an audio interface. The initial state is in the first setting, (number `0`).
- Alternate settings are often used to control the use of periodic endpoints, such as by having different endpoints use different amounts of reserved USB bandwidth. All standards-compliant USB devices that use isochronous endpoints will use them in non-default settings.

The `usb_interface` structure (1)

USB interfaces are represented by the `usb_interface` structure. It is what the USB core passes to USB drivers.

> `struct usb_host_interface *altsetting;`

List of alternate settings that may be selected for this interface, in no particular order.

The `usb_host_interface` structure for each alternate setting allows to access the `usb_endpoint_descriptor` structure for each of its endpoints:
`interface->alsetting[i]->endpoint[j]->desc`

> `unsigned int num_altsetting;`

The number of alternate settings.

The `usb_interface` structure (2)

- `struct usb_host_interface *cur_altsetting;`

  The currently active alternate setting.

- `int minor;`

  Minor number this interface is bound to.

  (for drivers using `usb_register_dev()`, described later).

Other fields in the structure shouldn't be needed by USB drivers.

Configurations

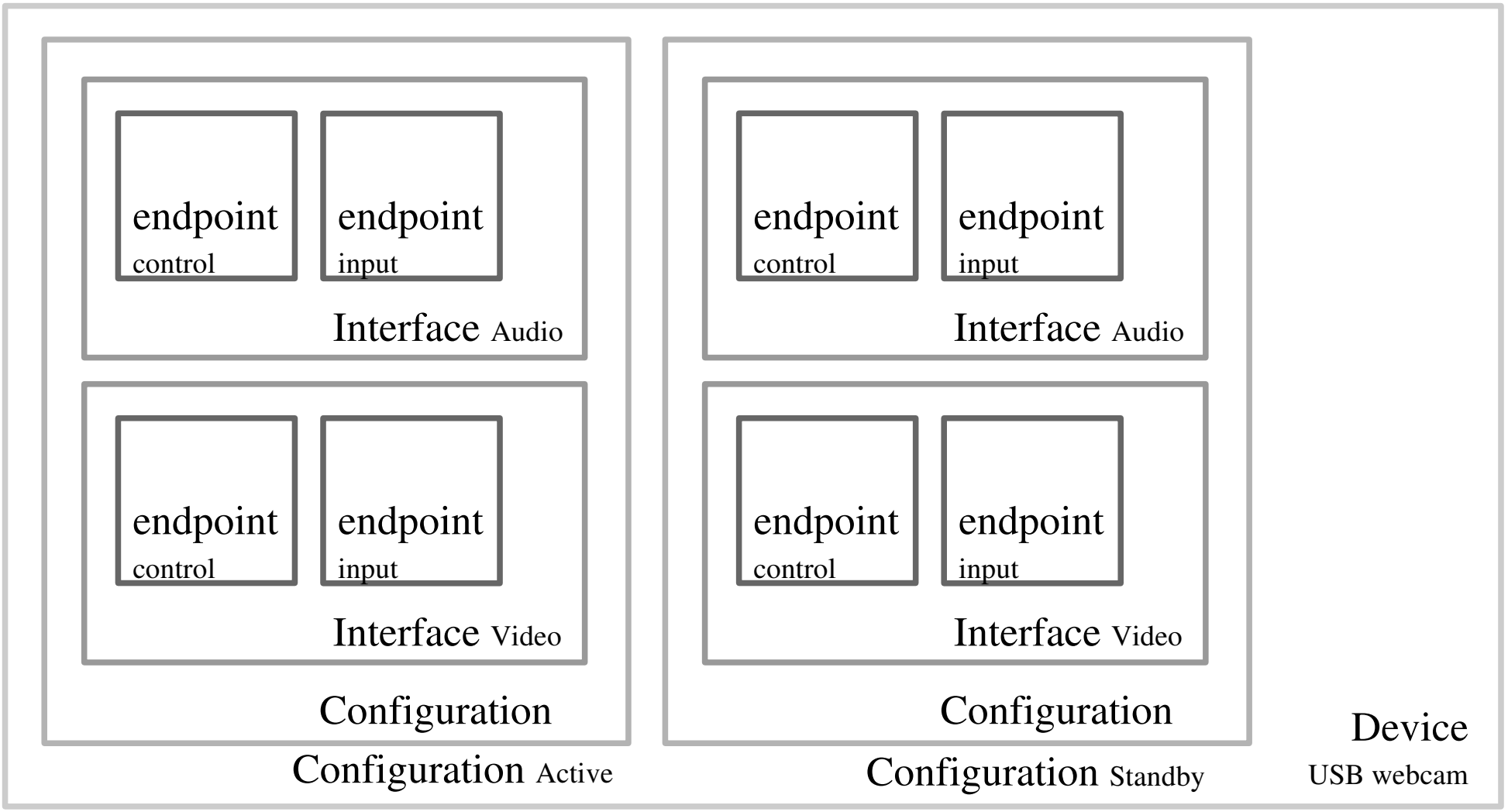
Interfaces are bundled into configurations.

- Configurations represent the state of the device.
  Examples: Active, Standby, Initialization
- Configurations are described
  with the `usb_host_config` structure.
- However, drivers do not need to access this structure.

Devices

- Devices are represented by the `usb_device` structure.
- We will see later that several USB API functions need such a structure.
- Many drivers use the `interface_to_usbdev()` function to access their `usb_device` structure from the `usb_interface` structure they are given by the USB core.

USB device overview



USB devices - Summary

- Hierarchy: device → configurations → interfaces → endpoints
- 4 different types of endpoints
  - control: device control, accessing information, small transfers. Guaranteed bandwidth.
  - interrupt (keyboards, mice...): data transfer at a fixed rate. Guaranteed bandwidth.
  - bulk (storage, network, printers...): use all remaining bandwidth. No bandwidth or latency guarantee.
  - isochronous (audio, video...): guaranteed speed. Possible data loss.

Linux USB basics

User-space representation

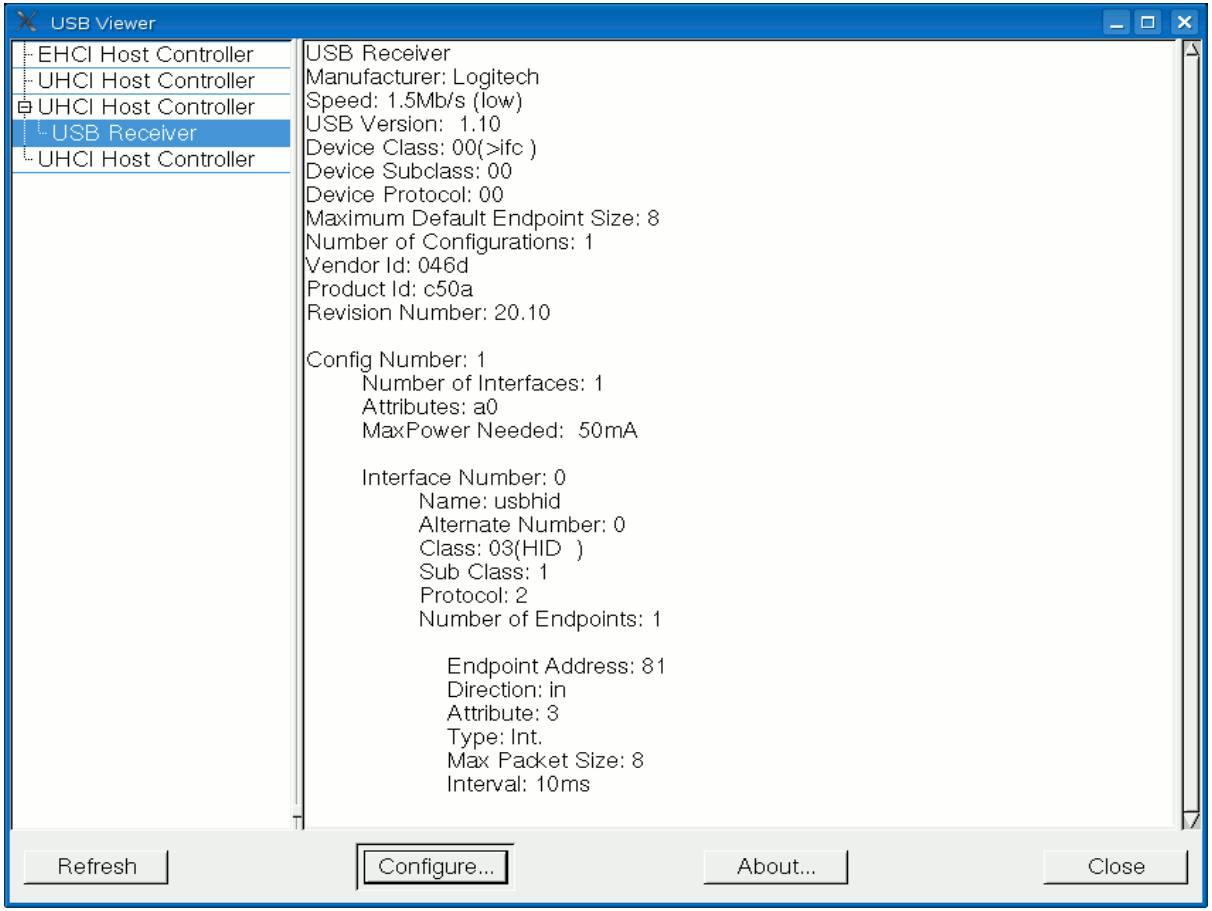
usbview

http://usbview.sourceforge.net

Graphical display

of the contents of

`/proc/bus/usb/devices`.



usbtree

http://www.linux-usb.org/usbtree

Also displays information from `/proc/bus/usb/devices`:

```
> usbtree
\/:  Bus 04.Port 1: Dev 1, Class=root_hub,
Driver=ehci_hcd/6p, 480M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p,
12M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p,
12M
    |__ Port 1: Dev 7, If 0, Class=HID, Driver=usbhid, 1.5M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=uhci_hcd/2p,
12M
```

Linux USB communication

USB Request Blocks

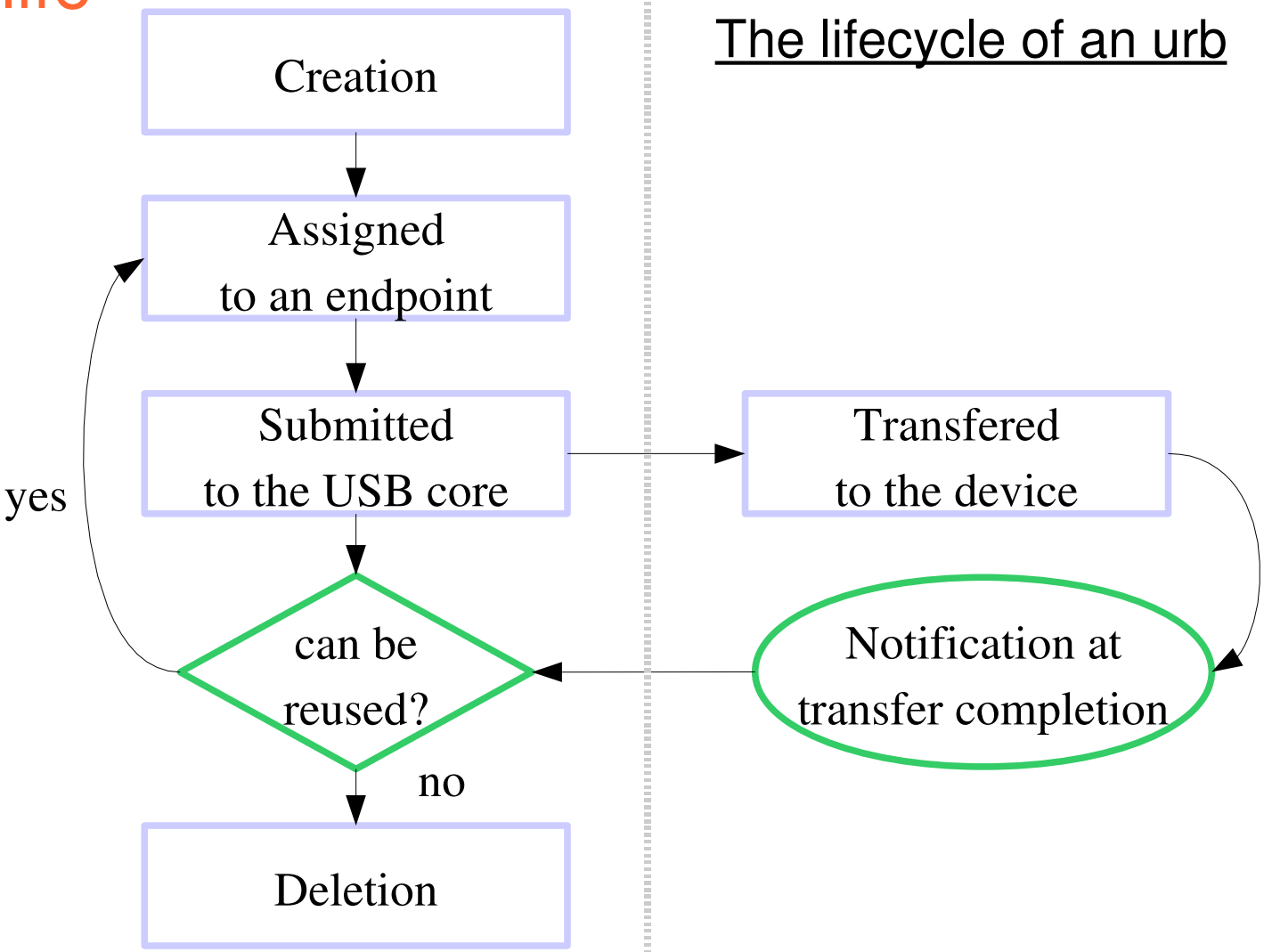
USB Request Blocks

- Any communication between the host and device is done asynchronously using USB Request Blocks (urbs).
- They are similar to packets in network communications.
- Every endpoint can handle a queue of urbs.
- Every urb has a completion handler.
- A driver may allocate many urbs for a single endpoint, or reuse the same urb for different endpoints.

See `Documentation/usb/URB.txt` in kernel sources.

Urban life

Device driver



The `urb` structure (1)

Fields of the `urb` structure useful to USB device drivers:

- `struct usb_device *dev;`

  Device the urb is sent to.

- `unsigned int pipe;`

  Information about the endpoint in the target device.

- `int status;`

  Transfer status.

- `unsigned int transfer_flags;`

  Instructions for handling the urb.

The `urb` structure (2)

- `void * transfer_buffer;`
  Buffer storing transferred data.
  Must be created with `kmalloc()`!
- `dma_addr_t transfer_dma;`
  Data transfer buffer when DMA is used.
- `int transfer_buffer_length;`
  Transfer buffer length.
- `int actual_length;`
  Actual length of data received or sent by the urb.
- `usb_complete_t complete;`
  Completion handler called when the transfer is complete.

The `urb` structure (3)

> `void *context;`

Data blob which can be used in the completion handler.

> `unsigned char *setup_packet;` (control urbs)

Setup packet transferred before the data in the transfer buffer.

> `dma_addr_t setup_dma;` (control urbs)

Same, but when the setup packet is transferred with DMA.

> `int interval;` (isochronous and interrupt urbs)

Urb polling interval.

> `int error_count;` (isochronous urbs)

Number of isochronous transfers which reported an error.

The `urb` structure (4)

> `int start_frame;` (isochronous urbs)

Sets or returns the initial frame number to use.

> `int number_of_packets;` (isochronous urbs)

Number of isochronous transfer buffers to use.

> `struct usb_iso_packet_descriptor` (isochronous urbs)

`iso_frame_desc[0];`

Allows a single urb to define multiple isochronous transfers at once.

Creating pipes

Functions used to initialize the `pipe` field of the `urb` structure:

- Control pipes
  `usb_sndctrlpipe(), usb_rcvctrlpipe()`
- Bulk pipes
  `usb_sndbulkpipe(), usb_rcvbulkpipe()`
- Interrupt pipes
  `usb_sndintpipe(), usb_rcvintpipe()`
- Isochronous pipes
  `usb_sndisocpipe(), usb_rcvisocpipe()`

Prototype

send (out) receive (in)

```
unsigned int usb_[snd/rcv][ctrl/bulk/int/isoc]pipe(
        struct usb_device *dev, unsigned int endpoint);
```

Creating urbs

- `urb` structures must always be allocated with the `usb_alloc_urb()` function.

  That's needed for reference counting used by the USB core.

```
#include <linux/usb.h>

struct urb *usb_alloc_urb(
      int iso_packets,     // Number of isochronous
                           // packets the urb should contain.
                           // 0 for other transfer types
      gfp_t mem_flags);    // Standard kmalloc() flags
```

- Check that it didn't return `NULL` (allocation failed)!
- Typical example:

```
urb = usb_alloc_urb(0, GFP_KERNEL);
```

Freeing urbs

> Similarly, you have to use a dedicated function to release urbs:

```
void usb_free_urb(struct urb *urb);
```

USB Request Blocks - Summary

- Basic data structure used in any USB communication.
- Implemented by the `struct urb` type.
- Must be created with the `usb_alloc_urb()` function. Shouldn't be allocated statically or with `kmalloc()`.
- Must be deleted with `usb_free_urb()`.

Linux USB communication

Initializing and submitting urbs

Initializing interrupt urbs

```
void usb_fill_int_urb (
   struct urb *urb,                  // urb to be initialized
   struct usb_device *dev,     // device to send the urb to
   unsigned int pipe,          // pipe (endpoint and device specific)
   void *transfer_buffer, // transfer buffer
   int buffer_length,          // transfer buffer size
   usb_complete_t complete,    // completion handler
   void *context,              // context (for handler)
   int interval                      // Scheduling interval (see next
   page)
   );
```

- This doesn't prevent you from making more changes to the urb fields before urb submission.
- The `transfer_flags` field needs to be set by the driver.

urb scheduling interval

For interrupt and isochronous transfers

- Low-Speed and Full-Speed devices:
  the `interval` unit is frames (ms)
- Hi-Speed devices:
  the `interval` unit is microframes (1/8 ms)

Initializing bulk urbs

Same parameters as in usb_fill_int_urb(),

except that there is no interval parameter.

```
void usb_fill_bulk_urb (
   struct urb *urb,              // urb to be initialized
   struct usb_device *dev,       // device to send the urb to
   unsigned int pipe,            // pipe (endpoint and device specific)
   void *transfer_buffer,        // transfer buffer
   int buffer_length,            // transfer buffer size
   usb_complete_t complete,      // completion handler
   void *context,                // context (for handler)
   );
```

Initializing control urbs

Same parameters as in usb_fill_bulk_urb(),
except that there is a setup_packet parameter.

```
void usb_fill_control_urb (
   struct urb *urb,                  // urb to be initialized
   struct usb_device *dev,    // device to send the urb to
   unsigned int pipe,         // pipe (endpoint and device specific)
   unsigned char *setup_packet, // setup packet data
   void *transfer_buffer, // transfer buffer
   int buffer_length,         // transfer buffer size
   usb_complete_t complete,   // completion handler
   void *context,             // context (for handler)
   );
```

Note that many drivers use the usb_control_msg() function instead (explained later).

Initializing isochronous urbs

No helper function. Has to be done manually by the driver.

```
for (i=0; i < USBVIDEO_NUMSBUF; i++) {
    int j, k;
    struct urb *urb = uvd->sbuf[i].urb;
    urb->dev = dev;
    urb->context = uvd;
    urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp);
    urb->interval = 1;
    urb->transfer_flags = URB_ISO_ASAP;
    urb->transfer_buffer = uvd->sbuf[i].data;
    urb->complete = usbvideo_IsocIrq;
    urb->number_of_packets = FRAMES_PER_DESC;
    urb->transfer_buffer_length = uvd->iso_packet_len * FRAMES_PER_DESC;
    for (j=k=0; j < FRAMES_PER_DESC; j++, k += uvd->iso_packet_len) {
        urb->iso_frame_desc[j].offset = k;
        urb->iso_frame_desc[j].length = uvd->iso_packet_len;
    }
}
```

`drivers/media/video/usbvideo/usbvideo.c` example

Allocating DMA buffers (1)

You can use the `usb_buffer_alloc()` function to allocate a DMA consistent buffer:

```
void *usb_buffer_alloc (
   struct usb_device *dev, // device
   size_t size,            // buffer size
   gfp_t mem_flags,        // kmalloc() flags
   dma_addr_t *dma         // (output) DMA address
);                         // of the buffer.
```

Example:

```
buf = usb_buffer_alloc(dev->udev,
    count, GFP_KERNEL, &urb->transfer_dma);
```

Allocating DMA buffers (2)

- To use these buffers, use the `URB_NO_TRANSFER_DMA_MAP` or `URB_NO_SETUP_DMA_MAP` settings for `urb->transfer_flags` to indicate that `urb->transfer_dma` or `urb->setup_dma` are valid on submit.
- Examples:

```
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
u->transfer_flags |= URB_NO_SETUP_DMA_MAP;
```

- Freeing these buffers:

```
void usb_buffer_free (
    struct usb_device *dev, // device
    size_t size,            // buffer size
    void *addr,             // CPU address of buffer
    dma_addr_t dma          // DMA address of buffer
);
```

Submitting urbs

After creating and initializing the urb

```
int usb_submit_urb(
   struct urb *urb,          // urb to submit
   int mem_flags);           // kmalloc() flags
```

`mem_flags` is used for internal allocations performed by `usb_submit_urb()`. Settings that should be used:

- GFP_ATOMIC: called from code which cannot sleep: a urb completion handler, hard or soft interrupts. Or called when the caller holds a spinlock.
- GPF_NOIO: in some cases when block storage is used.
- GFP_KERNEL: in other cases.

usb_submit_urb return values

`usb_submit_urb()` immediately returns:

- `0`: Request queued
- `-ENOMEM`: Out of memory
- `-ENODEV`: Unplugged device
- `-EPIPE`: Stalled endpoint
- `-EAGAIN`: Too many queued ISO transfers
- `-EFBIG`: Too many requested ISO frames
- `-EINVAL`: Invalid INT interval
  More than one packet for INT

Canceling urbs asynchronously

To cancel a submitted urb without waiting

> `int usb_unlink_urb(struct urb *urb);`

> Success: returns -EINPROGRESS

> Failure: any other return value. It can happen:

- When the urb was never submitted
- When the has already been unlinked
- When the hardware is done with the urb,
  even if the completion handler hasn't been called yet.

> The corresponding completion handlers will still be run and will see `urb->status == -ECONNRESET.`

Canceling urbs synchronously

To cancel an urb and wait for all completion handlers to complete

- This guarantees that the urb is totally idle and can be reused.
- `void usb_kill_urb(struct urb *urb);`
- Typically used in a `disconnect()` callback or `close()` function.
- Caution: this routine mustn't be called in situations which can not sleep: in interrupt context, in a completion handler, or when holding a spinlock.

See comments in `drivers/usb/core/urb.c` in kernel sources for useful details.

Initializing and submitting urbs - Summary

- `urb` structure fields can be initialized with helper functions `usb_fill_int_urb()`, `usb_fill_bulk_urb()`, `usb_fill_control_urb()`
- Isochronous urbs have to be initialized by hand.
- The `transfer_flags` field must be initialized manually by each driver.
- Use the usb_submit_urb() function to queue urbs.
- Submitted urbs can be canceled using usb_unlink_urb() (asynchronous) or usb_kill_urb() (synchronous).

Linux USB communication

Completion handlers

When is the completion handler called?

The completion handler is called in interrupt context, in only 3 situations.

Check the error value in `urb->status`.

- After the data transfer successfully completed.
  `urb->status == 0`
- Error(s) happened during the transfer.
- The urb was unlinked by the USB core.

`urb->status` should only be checked from the completion handler!

Transfer status (1)

Described in `Documentation/usb/error-codes.txt`

The urb is no longer “linked” in the system

- `-ECONNRESET`
  The urb was unlinked by `usb_unlink_urb()`.
- `-ENOENT`
  The urb was stopped by `usb_kill_urb()`.
- `-ESHUTDOWN`
  Error in from the host controller driver. The device was disconnected from the system, the controller was disabled, or the configuration was changed while the urb was sent.
- `-ENODEV`
  Device removed. Often preceded by a burst of other errors, since the hub driver doesn't detect device removal events immediately.

Transfer status (2)

Typical hardware problems with the cable or the device (including its firmware)

> -EPROTO

Bitstuff error, no response packet received in time by the hardware, or unknown USB error.

> -EILSEQ

CRC error, no response packet received in time, or unknown USB error.

> -EOVERFLOW

The amount of data returned by the endpoint was greater than either the max packet size of the endpoint or the remaining buffer size. "Babble".

Transfer status (3)

Other error status values

- `-EINPROGRESS`
  Urb not completed yet. Your driver should never get this value.
- `-ETIMEDOUT`
  Usually reported by synchronous USB message functions
  when the specified timeout was exceed.
- `-EPIPE`
  Endpoint stalled. For non-control endpoints,
  reset this status with `usb_clear_halt()`.
- `-ECOMM`
  During an IN transfer, the host controller received data from an endpoint
  faster than it could be written to system memory.

Transfer status (4)

- `-ENOSR`

  During an OUT transfer, the host controller could not retrieve data from system memory fast enough to keep up with the USB data rate.

- `-EREMOTEIO`

  The data read from the endpoint did not fill the specified buffer, and `URB_SHORT_NOT_OK` was set in `urb->transfer_flags`.

- `-EXDEV`

  Isochronous transfer only partially completed.
  Look at individual frame status for details.

- `-EINVAL`

  Typically happens with an incorrect urb structure field or `usb_submit_urb()` function parameter.

Completion handler implementation

- Prototype:

```
void (*usb_complete_t)(
    struct urb *,      // The completed urb
    struct pt_regs *   // Register values at the time
                       // of the corresponding
interrupt (if any)
);
```

- Remember you are in interrupt context:
  - Do not execute call which may sleep (use GFP_ATOMIC, etc.).
  - Complete as quickly as possible.
    Schedule remaining work in a tasklet if needed.

Completion handler - Summary

- The completion handler is called in interrupt context.
  Don't run any code which could sleep!
- Check the `urb->status` value in this handler,
  and not before.
- Success: `urb->status == 0`
- Otherwise, error status described in
  `Documentation/usb/error-codes.txt`.

Writing USB drivers

Supported devices

What devices does the driver support?

Or what driver supports a given device?

- Information needed by user-space, to find the right driver to load or remove after a USB hotplug event.
- Information needed by the driver, to call the right `probe()` and `disconnect()` driver functions (see later).

Such information is declared in a `usb_device_id` structure by the driver `init()` function.

The `usb_device_id` structure (1)

Defined according to USB specifications and described in `include/linux/mod_devicetable.h`.

> `__u16 match_flags`

  Bitmask defining which fields in the structure are to be matched against. Usually set with helper functions described later.

> `__u16 idVendor, idProduct`

  USB vendor and product id, assigned by the USB-IF.

> `__u16 bcdDevice_lo, bcdDevice_hi`

  Product version range supported by the driver, expressed in binary-coded decimal (BCD) form.

The `usb_device_id` structure (2)

- `__u8 bDeviceClass, bDeviceSubClass, bDeviceProtocol`

  Class, subclass and protocol of the device.

  Numbers assigned by the USB-IF.

  Products may choose to implement classes, or be vendor-specific.

  Device classes specify the behavior of all the interfaces on a device.

- `__u8 bInterfaceClass, bInterfaceSubclass, bInterfaceProtocol`

  Class, subclass and protocol of the individual interface.

  Numbers assigned by the USB-IF.

  Interface classes only specify the behavior of a given interface.

  Other interfaces may support other classes.

The `usb_device_id` structure (3)

- `kernel_ulong_t  driver_info`

  Holds information used by the driver. Usually it holds a pointer to a descriptor understood by the driver, or perhaps device flags.

  This field is useful to differentiate different devices from each other in the probe() function.

Declaring supported devices (1)

`USB_DEVICE(vendor, product)`

- Creates a `usb_device_id` structure which can be used to match only the specified vendor and product ids.
- Used by most drivers for non-standard devices.

`USB_DEVICE_VER(vendor, product, lo, hi)`

- Similar, but only for a given version range.
- Only used 11 times throughout Linux 2.6.18!

Declaring supported devices (2)

`USB_DEVICE_INFO (class, subclass, protocol)`

> Matches a specific class of USB devices.

`USB_INTERFACE_INFO (class, subclass, protocol)`

> Matches a specific class of USB interfaces.

The above 2 macros are only used in the implementations of standard device and interface classes.

Declaring supported devices (3)

Created usb_device_id structures are declared with the MODULE_DEVICE_TABLE() macro as in the below example:

```
/* Example from drivers/usb/net/catc.c */
static struct usb_device_id catc_id_table [] = {
   { USB_DEVICE(0x0423, 0xa) }, /* CATC Netmate, Belkin F5U011 */
   { USB_DEVICE(0x0423, 0xc) }, /* CATC Netmate II, Belkin F5U111 */
   { USB_DEVICE(0x08d1, 0x1) }, /* smartBridges smartNIC */
   { }                          /* Terminating entry */
};

MODULE_DEVICE_TABLE(usb, catc_id_table);
```

Note that MODULE_DEVICE_TABLE() is also used with other subsystems: pci, pcmcia, serio, isapnp, input...

Supported devices - Summary

- Drivers need to announce the devices they support in `usb_device_id` structures.
- Needed for user space to know which module to (un)load, and for the kernel which driver code to execute, when a device is inserted or removed.
- Most drivers use `USB_DEVICE()` to create the structures.
- These structures are then registered with `MODULE_DEVICE_TABLE(usb, xxx).`

Writing USB drivers

Registering a USB driver

The `usb_driver` structure

USB drivers must define a `usb_driver` structure:

> `const char *name`

Unique driver name. Usually be set to the module name.

> `const struct usb_device_id *id_table;`

The table already declared with `MODULE_DEVICE_TABLE()`.

> `int (*probe) (struct usb_interface *intf, const struct usb_device_id *id);`

Probe callback (detailed later).

> `void (*disconnect) (struct usb_interface *intf);`

Disconnect callback (detailed later).

Optional `usb_driver` structure fields

- ```
  int (*suspend) (struct usb_interface *intf,
                  pm_message_t message);
  int (*resume) (struct usb_interface *intf);
  ```

  Power management: callbacks called before and after the USB core suspends and resumes the device.

- ```
  void (*pre_reset) (struct usb_interface *intf);
  void (*post_reset) (struct usb_interface *intf);
  ```

  Called by `usb_reset_composite_device()` before and after it performs a USB port reset.

Driver registration

Use usb_register() to register your driver. Example:

```
/* Example from drivers/usb/input/mtouchusb.c */

static struct usb_driver mtouchusb_driver = {
        .name           = "mtouchusb",
        .probe          = mtouchusb_probe,
        .disconnect     = mtouchusb_disconnect,
        .id_table       = mtouchusb_devices,
};

static int __init mtouchusb_init(void)
{
        dbg("%s - called", __FUNCTION__);
        return usb_register(&mtouchusb_driver);
}
```

Driver unregistration

Use usb_unregister() to register your driver. Example:

```
/* Example from drivers/usb/input/mtouchusb.c */

static void __exit mtouchusb_cleanup(void)

{
        dbg("%s - called", __FUNCTION__);
        usb_deregister(&mtouchusb_driver);
}
```

probe() and disconnect() functions

- The `probe()` function is called by the USB core to see if the driver is willing to manage a particular interface on a device.
- The driver should then make checks on the information passed to it about the device.
- If it decides to manage the interface, the `probe()` function will return `0`. Otherwise, it will return a negative value.
- The `disconnect()` function is called by the USB core when a driver should no longer control the device (even if the driver is still loaded), and should do some clean-up.

Context: USB hub kernel thread

- The `probe()` and `disconnect()` callbacks are called in the context of the USB hub kernel thread.
- So, it is legal to call functions which may sleep in these functions.
- However, all addition and removal of devices is managed by this single thread.
- Most of the probe function work should indeed be done when the device is actually opened by a user. This way, this doesn't impact the performance of the kernel thread in managing other devices.

probe() function work

- In this function the driver should initialize local structures which it may need to manage the device.
- In particular, it can take advantage of information it is given about the device.
- For example, drivers usually need to detect endpoint addresses and buffer sizes.

Time to show and explain examples in detail!

usb_set_intfdata() / usb_get_intfdata()

```
static inline void usb_set_intfdata (
      struct usb_interface *intf,
      void *data);
```

- Function used in `probe()` functions to attach collected device data to an interface. Any pointer will do!
- Useful to store information for each device supported by a driver, without having to keep a static data array.
- The `usb_get_intfdata()` function is typically used in the device open functions to retrieve the data.
- Stored data need to be freed in `disconnect()` functions:
  `usb_set_intfdata(interface, NULL);`

Plenty of examples are available in the kernel sources!

Writing USB drivers

USB transfers without URBs

Transfers without URBs

The kernel provides two `usb_bulk_msg()` and `usb_control_msg()` helper functions that make it possible to transfer simple bulk and control messages, without having to:

- Create or reuse an urb structure,
- Initialize it,
- Submit it,
- And wait for its completion handler.

Transfers without URBs - constraints

- These functions are synchronous and will make your code sleep. You must not call them from interrupt context or with a spinlock held.
- You cannot cancel your requests, as you have no handle on the URB used internally. Make sure your `disconnect()` function can wait for these functions to complete.

See the kernel sources for examples using these functions!

USB device drivers - Summary

Module loading

- Declare supported devices (interfaces).
- Bind them to `probe()` and `disconnect()` functions.

Supported devices are found

- `probe()` functions for matching interface drivers are called.
- They record interface information and register resources or services.

Devices are opened

- This calls data access functions registered by the driver.
- URBs are initialized.
- Once the transfers are over, completion functions are called. Data are copied from/to user-space.

Devices are removed

- The `disconnect()` functions are called.
- The drivers may be unloaded.

Advice for embedded system developers

If you need to develop a USB device driver for an embedded Linux system.

- Develop your driver on your GNU/Linux development host!
- The driver will run with no change on the target Linux system (provided you wrote portable code!): all USB device drivers are platform independent.
- Your driver will be much easier to develop on the host, because of its flexibility and the availability of debugging and development tools.

References

- Wikipedia's article on USB
  http://en.wikipedia.org/wiki/Universal_Serial_Bus
- The USB drivers chapter in the Linux Device Drivers book:
  http://lwn.net/Kernel/LDD3/ (Free License!)
- The Linux kernel sources (hundreds of examples, “Use the Source!”)
  Browse them with http://lxr.free-electrons.com.
- Linux USB project
  http://www.linux-usb.org/
- Linux kernel documentation:
  `Documentation/usb/`
  Linux USB API (generated from kernel sources):
  http://free-electrons.com/kerneldoc/latest/DocBook/usb/
- USB specifications:
  http://www.usb.org/developers/docs/