Développement de drivers pour Linux

EPITA 2021 – GISTRE
Geoffrey Le Gourriérec

Plan

- Petit tour dans le *sysfs*
- Les attributs

- Le rôle des classes

- La classe *misc*
- Créer sa classe à la main

Petit tour dans le *sysfs*

Rappel : *sysfs* vs *procfs*

- **procfs** est un **pseudo-système de fichiers** historique
  - Contient surtout des informations sur l'état courant du kernel et des processus
  - Outil obsolète aujourd'hui, pour un driver
- **sysfs** est aussi un pseudo-fs, plus récent
  - Vise à fournir des interfaces (lecture, écriture) entre l'espace utilisateur et le kernel
  - Ici, on s'intéresse aux répertoires représentant les devices

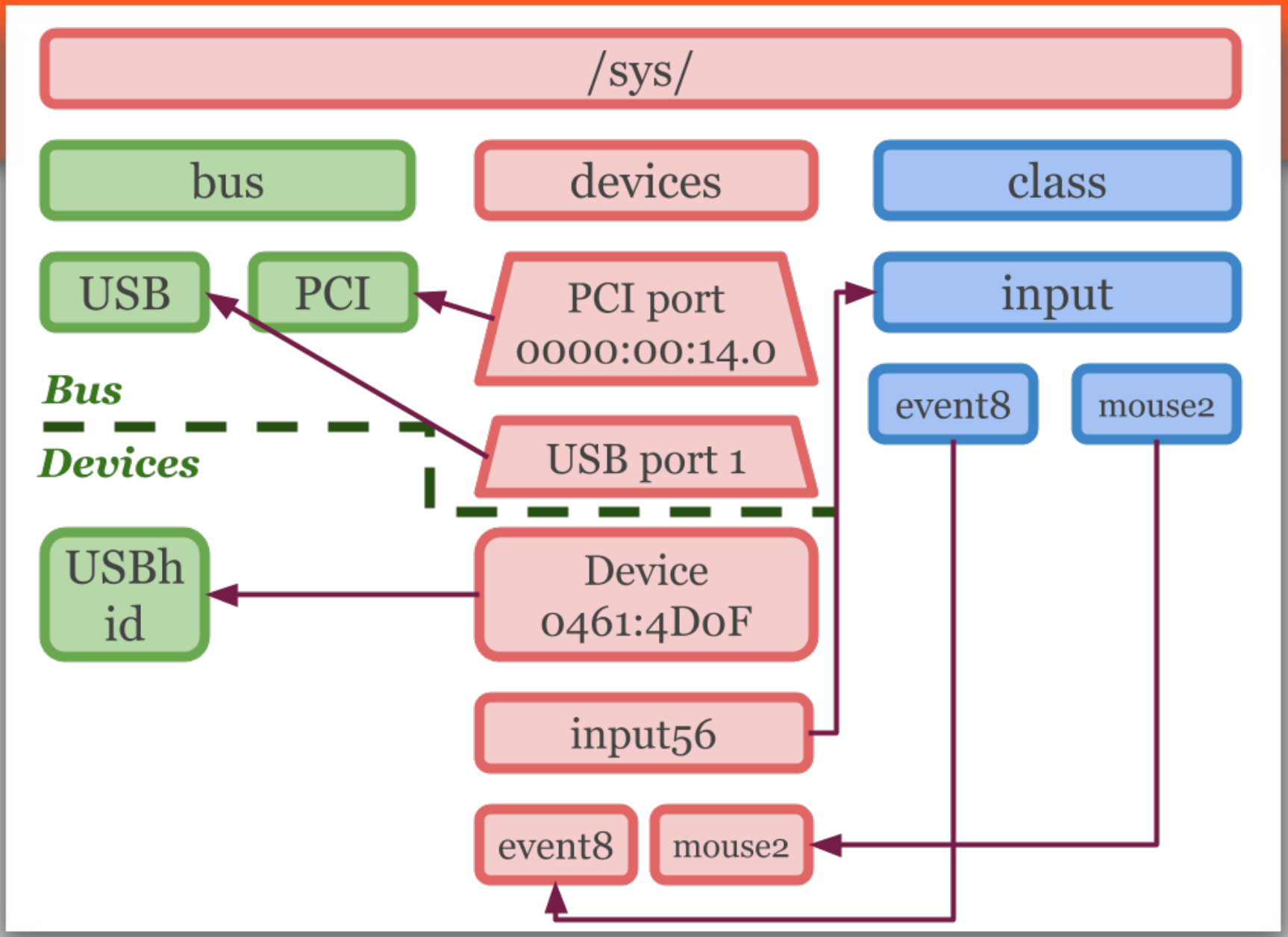
Petit tour dans le *sysfs*

Rappel : répertoires principaux du sysfs

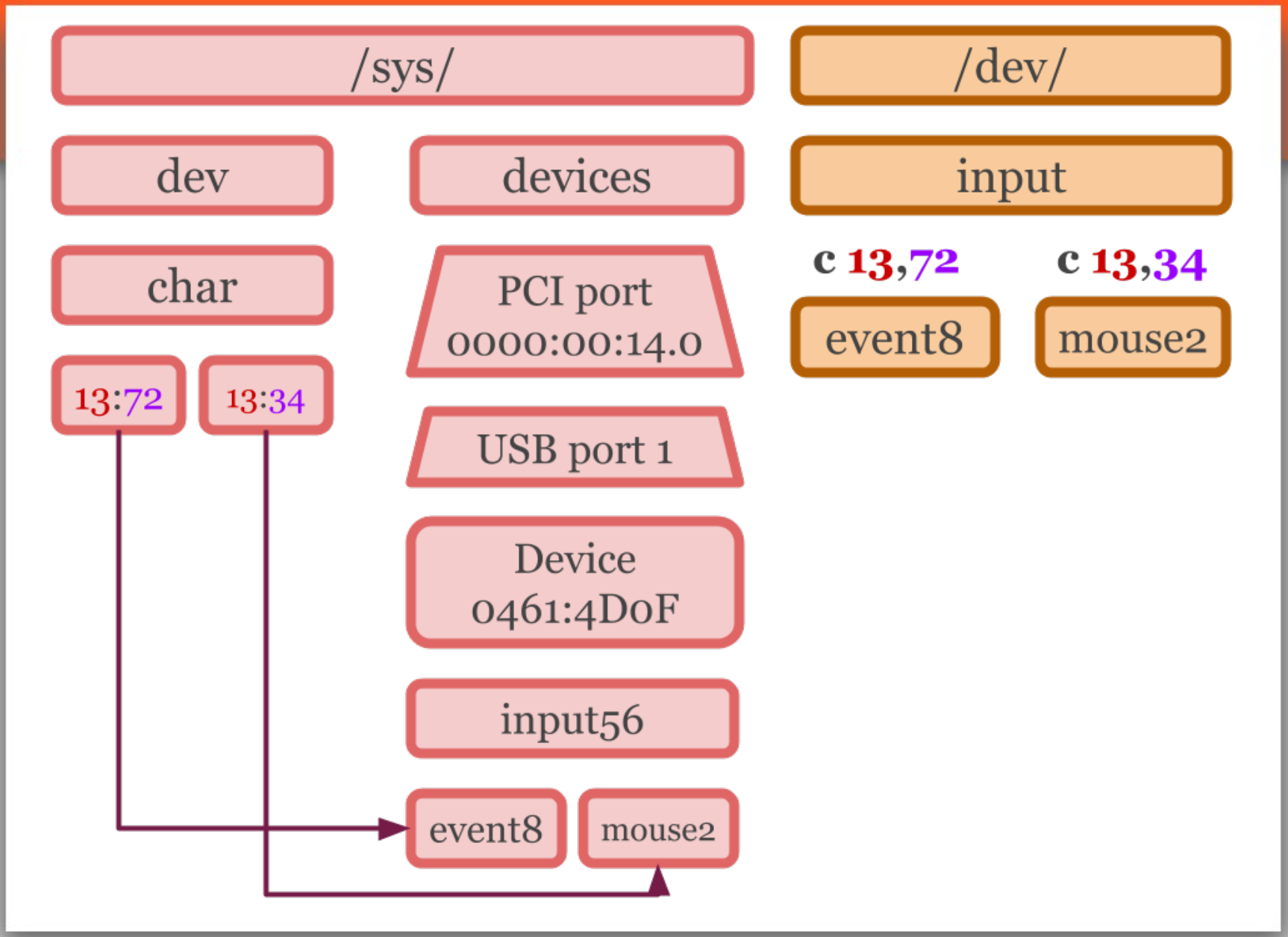
- *module/* : Tout ce qui concerne les bouts de code modularisés du kernel (actifs ou non)
- *dev/* : Devices organisés par majeur/mineur
- *devices/* : Devices rangés sans organisation particulière ; répertoire historique
- *bus/* : Devices organisés par bus (topologie)
- *class/* : Devices organisés par rôle fonctionnel

Prenons pour exemple une souris USB.

Petit tour dans le *sysfs*



Petit tour dans le *sysfs*



Les attributs

- Les fichiers dans */sys/devices*, */sys/bus*. . . sont des **attributs**
- Ils représentent une valeur particulière qui influence le comportement du driver
- La structure de base (dans *linux/sysfs.h*) est :

```
struct attribute {
    const char *name;
    umode_t     mode;
};
```

Les attributs

device_attribute

- *struct attribute* en elle-même est un peu vide…
- Le kernel en dérive quelques variantes, qui vont notamment apporter :
  - Une **fonction d'écriture** : *show()*
  - Une **fonction de lecture** : *store()*
- **Pour les devices**, *linux/device.h* définit :

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

Les attributs

- Les attributs sont rassemblés par **groupes**
- Sémantiquement, un groupe est censé réunir des attributs similaires
- *is_visible()* permet de cacher certains attributs selon une logique propre au driver

```
// Version simplifiée
struct attribute_group {
    const char          *name;
    umode_t             (*is_visible)(
        struct kobject *,
        struct attribute *,
        int);
    // Un bête tableau d'attributs
    struct attribute **attrs;
};
```

Les attributs

Exemple concret : *drivers/gpio/gpiolib-sysfs.c*

- *edge, direction, value, active_low*
- Utilisation de *DEVICE_ATTR_RO(), DEVICE_ATTR_RW()* pour définir les attributs, avec leurs fonctions *show()* et *store()*
- *gpio_groups* regroupe l'**unique groupe d'attributs**
- Ensuite utilisé dans *gpiod_export()* , qui est invoquée par l'**attribut de classe** "export"
  - Nous verrons ces “classes” juste après
- Cet exemple est également intéressant pour la suite, alors n’hésitez pas à retourner l’examiner :)

Le rôle des classes

Organiser par fonction

- *sys/class/* ne compte quasiment que des liens symboliques vers */sys/devices/*
- Permet au userland de **raisonner par abtraction**, sans connaître la topologie (bus) associée au device
- Udev est un contre-exemple : il cherche à parcourir la topologie sous-jacente pour propager des évènements
- Donner une **interface sysfs commune**
  - Comme */sys/bus/* ou */sys/devices/* , chaque sous-répertoire peut contenir des attributs
  - Ici, le but est d'en créer des communs, qui reprennent un besoin partagé

Le rôle des classes

Petits points de confusion

- Une classe est une manière d'abstraire des détails de manipulation entre devices
- Exemple : un disque SCSI et un disque ATA
- Mais elles ne **permettent pas de factoriser du code "de driver"** (qui touche au HW)
- C'est aux développeurs de drivers d'éventuellement établir des couches de généricité
- **Classe n'implique pas driver en mode caractère**
- **N'importe-quel module** peut déclarer / utiliser une classe

Le rôle des classes

Où sont définies les classes ?

- **Chaque driver peut définir sa classe**
- En général, on implémente la classe dans un module générique, réutilisable par les “vrais” drivers
- Le module associé est chargé avec celui s'occupant de la "vraie" implémentation
- *drivers/gpio/gpiolib-sysfs.c* est un exemple de couche "générique"
  - Deux attributs de classe : "export" et "unexport"
  - Les autres attributs sont, eux, spécifiques au device manipulé
  - Votre driver GPIO s’occupera toujours des détails

La classe *misc*

- **Une classe prémâchée** pour les drivers en mode caractère
- **Evite de devoir écrire un driver complet** quand on n'a besoin que d'1 ou 2 fonctions
- Une seule structure fait tout :

```
struct miscdevice {
    int minor;
    const struct file_operations  *fops;
    struct list_head               list;
    struct device                 *parent;
    struct device                 *this_device;
    const struct attribute_group **groups;
    const char                    *nodename;
    umode_t                        mode;
};
```

La classe *misc*

Particularités

- **Le majeur est toujours 10**
  - Le mineur (unique) était historiquement assigné par la LANANA (*Linux Assigned Names And Numbers Authority*)
  - Maintenant sous la coupelle *Linux Standard Base*
- Créé un **device virtuel**
  - Créé notre classe dans */sys/class/misc/*
  - Créé notre device dans */sys/devices/virtual/misc/*
- Rattaché automatiquement à udev
  - Un noeud va être automatiquement créé dans */dev/*
  - Il sera également automatiquement détruit

La classe *misc*

Fonctions

- *misc_register()* créé le device dans le sysfs
- *misc_unregister()* le détruit

```
int  misc_register(struct miscdevice *misc);
void misc_unregister(struct miscdevice *misc);
```

La classe *misc*

Exercice : pingpong, le retour

- Réimplémenter "ping-pong", mais avec la classe misc
- Observer les différences
  - Dans le code
  - Dans *sysfs*, *devfs*

Créer sa classe à la main

- *Misc*, c'est bien, mais ce n'est pas propice aux extensions futures de votre driver
- On cherche plutôt à réutiliser une classe qui fasse sens par rapport au *device* (e.g. "input" pour claviers, souris…)
- S'il n'y en a pas, créons la nôtre !

- Dans la vaste majorité des cas, on n'a pas besoin d'attributs exclusifs à la classe (gpio est en fait un cas rare)
  - Dans ce cas, *class_create()* est utilisée

```
struct class *class_create(
   struct module *owner,
   const char    *name);
void class_destroy(
   struct class *cls);
```

Créer sa classe à la main

- Une classe, c'est le répertoire parent; chaque driver va devoir créer le sien, à présent
- Les devices virtuels sous le répertoire sysfs de notre classe sont créés avec *device_create()*
- Comme mentionné précédemment, pas moyen d'associer juste 1 attribut directement

```
struct device *device_create(
   struct class *cls,
   struct device *parent,
   dev_t devt,
   void *drvdata,
   const char *fmt, ...);
void device_destroy(
   struct class *cls,
   dev_t devt);
```

Créer sa classe à la main

- Les attributs auront besoin de fonctions *show()* et *store()*
- Les variantes tournent autour des droits d'accès: lecture / écriture
  - Naturellement, un attribut *RO* n'a pas besoin de la fonction *store()*

```
ssize_t my_attr_show(
   struct device *dev,
   struct class_attribute *attr,
   char *buf);
ssize_t my_attr_store(
   struct device *dev,
   struct class_attribute *attr,
   char *buf,
   size_t size);
```

Créer sa classe à la main

- Définir les fonctions ne suffit pas: il faut les *exporter* via des macros standards
- Attention au nommage: définir un attribut *foo* fait chercher “en dur” les fonctions *foo_show()* et *foo_store()*

```
// Déf. cherchant my_dev_attr_{show|store}
DEVICE_ATTR_{RO|WO|RW}(my_dev_attr);
```

Créer sa classe à la main

Les groupes d'attributs

- Encore un niveau de *is_visible()*
- Sinon, juste un tableau d'attributs…
- Ici, faire un groupe d'un seul attribut "foo":

```
static struct attribute *my_attrs[] = {
    &dev_attr_foo.attr,
    // Ne pas oublier NULL à la fin
    NULL
};
static const struct attribute_group my_grp = {
    .attrs      = my_attrs,
    .is_visible = my_grp_is_visible,
};
static const struct attribute_group *my_grps[] = {
    &my_grp,
    NULL
};
```

Créer sa classe à la main

On récapitule…

- On peut définir un ou plusieurs attributs
  - Droits (lecture/écriture) propres
  - Simple buffer dans lequel lire/écrire
  - A nous de définir le contenu
- Ces attributs sont obligatoirement *groupés*
- Votre driver donne obligatoirement au kernel une *liste de groupes d'attributs*

Créer sa classe à la main

Et au sommet de la pyramide…

- *device_create_with_groups()* est à utiliser dès que l'on souhaite ajouter des attributs à son driver
- Même destruction par *device_destroy()*
- A appeler avant *cdev_add()*

```
struct device *device_create_with_groups(
   struct class *cls,
   struct device *parent,
   dev_t devt,
   void *drvdata,
   struct attribute_group **attr_grps,
   const char *fmt, ...);
```

Créer sa classe à la main

Exercice : statistiques

- Un driver qui compte le nombre de fois que *read()* et *write()* ont été invoquées dans votre driver
- Un attribut en lecture seule par fonction

```
$> dd bs=1 count=1 if=/dev/stats0 of=/dev/null
$> cat /sys/class/statistics/stats0/nb_reads
1

$> dd bs=1 count=5 if=/dev/zero of=/dev/stats0
$> cat /sys/class/statistics/stats0/nb_writes
5
```

Créer sa classe à la main

Encore plus loin: les attributs de classe

- Admettons que nous voulions implémenter un **comportement partagé** par tous les drivers de notre classe (comme pour gpiolib-sysfs)
- On peut encore utiliser un autre type d'attributs, qui sont rattachés à la classe, et à aucun driver en particulier
- Sensiblement la même chose que ce qui précède
  - Toujours *show()* et *store()*
  - Toujours besoin de mettre en groupes

Créer sa classe à la main

Façonner sa classe

- On va créer sa classe vraiment à la main, ce qui nous permet d'ajouter les attributs dans la structure
- Définition de nos attributs de classe avec une deuxième macro

```
// L'instance de struct class vit dans
// le code du module qui la définit
int class_register(struct class *cls);
void class_unregister(struct class *cls);

// Déf. cherchant my_cls_attr_{show|store}
CLASS_ATTR_{RO|RW}(my_cls_attr);
```

Créer sa classe à la main

Exercice : statistiques+

- On reprend le driver précédent en ajoutant deux attributs de classe:
  - Un qui fait la moyenne des *read()*
  - L'autre qui fait la moyenne des *write()*
- D'abord s'assurer que votre driver sait gérer plusieurs *devices* :)

```
$> dd bs=1 count=5  if=/dev/zero of=/dev/stats0
$> dd bs=1 count=15 if=/dev/zero of=/dev/stats1
$> cat /sys/class/statistics/avg_nb_writes
10
```