Mécanismes de débogage

M2 SETI B4 / MS SE SE758

Guillaume Duc
guillaume.duc@telecom-paris.fr
2020–2021

Introduction

- Débogage plus complexe qu'une application « classique »
- Plusieurs mécanismes sont disponibles

Afficher des messages

Fonctions pr_*

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

pr_emerg(const char *fmt, ...);
pr_alert(const char *fmt, ...);
pr_crit(const char *fmt, ...);
pr_err(const char *fmt, ...);
pr_warning(const char *fmt, ...);
pr_notice(const char *fmt, ...);
pr_info(const char *fmt, ...);

pr_debug(const char *fmt, ...);
```

Afficher des messages

Fonctions `pr_*`

- Fonctionnent à la manière de la fonction `printf` classique
- Arguments
  - `const char *fmt` : chaîne de caractères à afficher incluant éventuellement des instructions de formatage (ex. `%d`, `%x`, `%s`...)
  - `...` : arguments supplémentaires si nécessaire
- Le message est stocké dans un tampon circulaire du noyau (`__log_buf`) consultable par la commande `dmesg`
- Si le niveau de priorité du message dépasse un certain seuil, le message est également affiché directement sur les consoles ouvertes
- L'appel à `pr_debug` est supprimé à la compilation si le symbole `DEBUG` n'est pas défini

Afficher des messages

Variantes dev_*

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

dev_emerg(const struct device *dev, const char *fmt, ...);
dev_alert(const struct device *dev, const char *fmt, ...);
dev_crit(const struct device *dev, const char *fmt, ...);
dev_err(const struct device *dev, const char *fmt, ...);
dev_warn(const struct device *dev, const char *fmt, ...);
dev_notice(const struct device *dev, const char *fmt, ...);
dev_info(const struct device *dev, const char *fmt, ...);

dev_dbg(const struct device *dev, const char *fmt, ...);
```

- Même fonctionnement que les fonctions `pr_*`
- Arguments supplémentaire `dev` : pointeur vers la structure `struct device` représentant votre périphérique
- Permet d'afficher le périphérique concerné en plus du message

Debugfs (GPL uniquement)
Présentation

- Système de fichier virtuel permettant de visualiser des variables internes de votre module depuis l'espace utilisateur
- Option de configuration `CONFIG_DEBUG_FS`
- Montage via `mount -t debugfs none /sys/kernel/debug`
- Documentation complète dans `Documentation/filesystems/debugfs.txt`

Debugfs (GPL uniquement)
Création d'un répertoire

```
#include <linux/debugfs.h>
struct dentry *debugfs_create_dir(const char *name,
                                  struct dentry *parent);
```

- Crée un répertoire de nom `name` sous le répertoire `parent` (ou `NULL` pour la racine de *debugfs*)

Debugfs (GPL uniquement)

Création de fichiers

```
struct dentry *debugfs_create_u8(const char *name, umode_t mode,
                                 struct dentry *parent, u8 *value);
struct dentry *debugfs_create_u16(const char *name, umode_t mode,
                                  struct dentry *parent, u16 *value);
struct dentry *debugfs_create_u32(const char *name, umode_t mode,
                                  struct dentry *parent, u32 *value);
struct dentry *debugfs_create_u64(const char *name, umode_t mode,
                                  struct dentry *parent, u64 *value);
```

- Crée un fichier de nom `name` ayant les permissions `mode`, dans le répertoire `parent` et reflétant la valeur de la variable pointée par `value`
- Représentation sous forme décimale (hexadécimale pour les fonctions `debugfs_create_x8`, `_x16`...)
- Si l'écriture est autorisée, il est possible de modifier dynamiquement `value` en écrivant dans ce fichier

Debugfs (GPL uniquement)

Création de fichiers

```
struct debugfs_blob_wrapper {
  void *data;
  unsigned long size;
};

struct dentry *debugfs_create_blob(const char *name, umode_t mode,
                                   struct dentry *parent,
                                   struct debugfs_blob_wrapper *blob);
```

- Exporte (en lecture seule) un bloc de données de taille arbitraire
- D'autres fonctions plus génériques sont disponibles (cf. doc)

Autres mécanismes

- `kgdb` : mécanisme, intégré au noyau (non compilé par défaut) se comportant comme un serveur `gdb`, permettant ainsi à un `gdb` tournant sur une autre machine, reliée par une liaison série, de se connecter au noyau et de le déboguer
- `JTAG/SWD` : connexion directe au processeur
- `QEMU` : émulation d'une machine et de ses périphériques, offre une interface avec `gdb` pour déboguer