Informatique embarquée 12/16

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

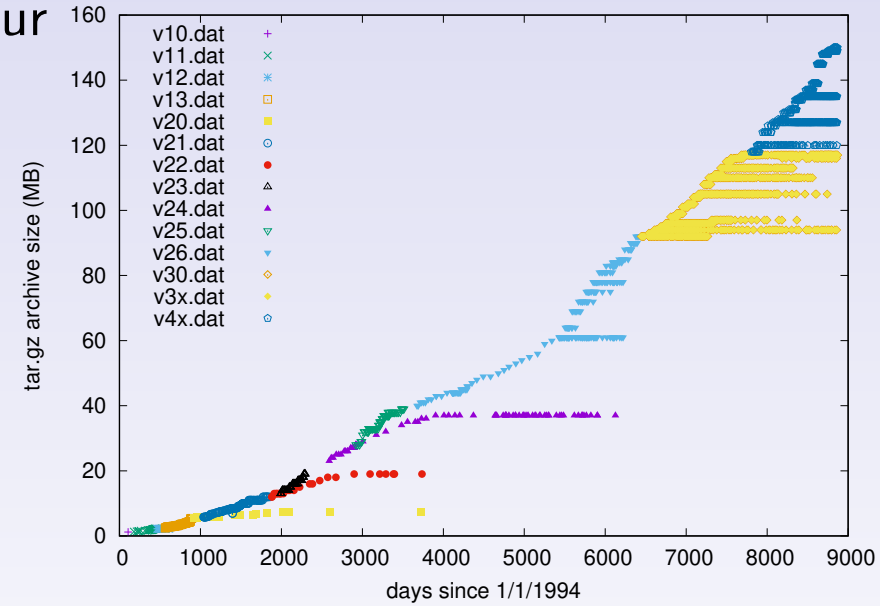
18 mars 2018

Introduction

- Développeur $\Rightarrow$ s'intéresser à l'interface matériel-utilisateur
- Seul le noyau a accès à toutes les ressources de la machine
- Distribuer les ressources et s'assurer de la cohérence des accès
- Ajout de fonctionnalités au noyau Linux : les modules

Porter GNU/Linux à une nouvelle architecture

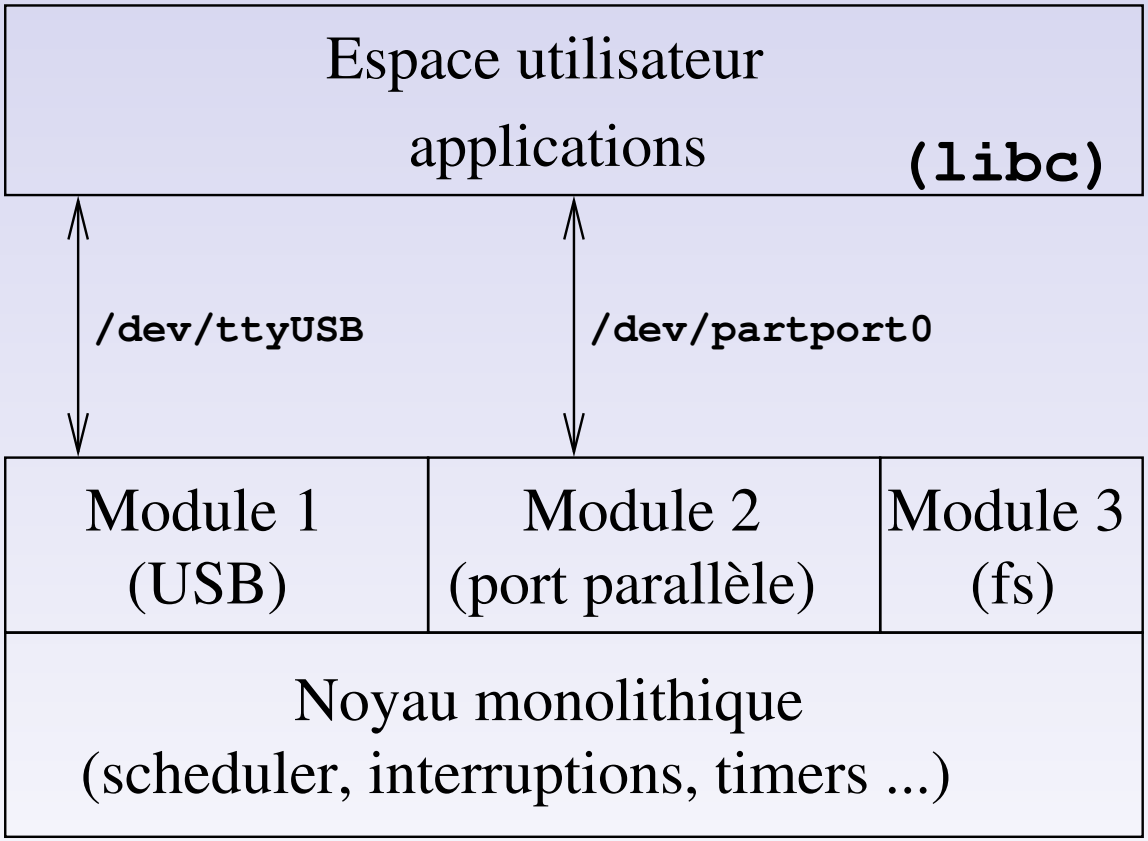
- une chaîne de compilation
- tenir compte de l'architecture mémoire de la cible (option -T de `ld`)
- développer un bootloader (initialisation des périphériques et chargement du noyau en mémoire) [1]
- développer les modules noyau pour supporter le matériel spécifique à la nouvelle architecture [2].



Évolution de la taille de l'archive `.tar.gz` du noyau linux (`www.kernel.org`)

1. S. Guinot, J.-M Friedt, *GNU/Linux sur Playstation Portable*, GNU/Linux Magazine France **114**, Mars 2009, pp.30-40
2. J. Corbet, A. Rubini & G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*, O'Reilly (2005), disponible à `http://lwn.net/Kernel/LDD3/`

Architecture d'un OS



fonctions système, réseau, gestion processus et mémoire, vfs

Le répertoire /dev

- contient des pseudo-fichiers faisant le lien entre un module noyau et l'espace utilisateur
- chaque *device* est défini par un *major number* et, dans sa classe, un *minor number*
- du côté du noyau, un pilote (*driver*) s'est lié au même major number et fournit des méthodes standard
  - `open()`
  - `close()`
  - `write()`
  - `read()`
  - `ioctl()`

```
static struct file_operations qadc_fops =
{
        owner:          THIS_MODULE,
        read:           qadc_read,
        open:           qadc_open,
        ioctl:          qadc_ioctl,
        release:        qadc_release,
};

static int __init qadc_init (void)
{...
 register_chrdev (qadc_major, "ppp", &qadc_fops);
 ...}

module_init (qadc_init);
module_exit (qadc_exit);
```

Structure de base

- deux points d'entrée et de sortie :
  `module_init (fonction_debut);` et
  `module_exit (fonction_fin);`
- initialisation du point de liaison avec l'espace utilisateur
  `register_chrdev (90 , "jmf" ,&fops );`
  attention au message d'erreur si le major est déjà occupé : cf `/proc/devices`
- des affichages au travers de `/var/log/syslog` par
  `printk (KERN_ALERT "message formaté");` (*sans virgule !*)
- possibilité de création dynamique de l'entrée dev

```
int hello_start()  // init_module(void)
{jmfdev.name = "jmf";  // /dev/jmf: major = classe misc
 jmfdev.minor = MISC_DYNAMIC_MINOR;
 jmfdev.fops = &fops;
 misc_register(&jmfdev);  // creation dynamique /dev/jmf
}
```

Fonctions de base

- `printf(char *)` (stdio) → `printk(level char*)` (`linux/kern_levels.h`)
- `malloc` → `kalloc()`
- MMU : `copy_from_user` & `copy_to_user` pour des blocs,
- MMU : `put_user` & get_user pour des scalaires
- Calculs flottants **interdits** :
  - pas d'affichage par `printk` (`%f`),
  - problème de sauvegarde du contexte du FPU,
  - inefficace en l'absence de FPU,
  - absence de `math.h` (bibliothèque en espace utilisateur)

Méthodes fournies par le noyau

Nombreuses fonctionnalités fournies par le noyau pour la gestion des tâches (ordonnanceur), *timer* ou communication avec l'espace utilisateur.

- concepts se rapprochant de la programmation microcontrôleur

```
init_timer_on_stack (&exp_timer );
exp_timer.expires = jiffies+delay*HZ ;
exp_timer.data = 0;
exp_timer.function = do_something ;
add_timer (&exp_timer);
```

- tâches (*tasklet*) : interruptions logicielles

```
char my_tasklet_data[]="my_tasklet_function was called";
void my_tasklet_function( unsigned long data )
{ printk( "%s\n", (char *)data ); return; }

DECLARE_TASKLET( my_tasklet, my_tasklet_function,
 (unsigned long) &my_tasklet_data );

int init_module( void )
{ tasklet_schedule( &my_tasklet ); return 0; }

void cleanup_module( void )
{ tasklet_kill( &my_tasklet ); return; }
```

→ ISR est brève et appelle une tasklet

Compilation d'un module noyau

- Nécessité de connaître la version et la configuration du noyau courant : `uname -a`
- Nécessité de connaître l'emplacement des sources du noyau : `/usr/src/linux-headers-ver-arch` (Debian), ou `$BUILDROOT/output/build/linux*` (ARM)
- Compilation du noyau Linux : `make modules` pour conclure
- Notre module (PC) :

```
obj-m +=0mymod.o 1mymod.o

all:
	make -C /usr/src/linux-headers-$(shell uname -r)/build \
	      M=$(PWD) modules
```

Compilation d'un module noyau

- Nécessité de connaître la version et la configuration du noyau courant : `uname -a`
- Nécessité de connaître l'emplacement des sources du noyau : `/usr/src/linux-headers-ver-arch` (Debian), ou `$BUILDROOT/output/build/linux*` (ARM)
- Compilation du noyau Linux : `make modules` pour conclure
- Notre module (Redpitaya) :

```
PATH:=$(PATH):$(HOME)/buildroot/output/host/usr/bin/

obj-m +=0mymod.o 1mymod.o

all:
	make ARCH=arm \
	     CROSS_COMPILE=arm-buildroot-linux-uclibcgnueabihf- \
	     -C $(HOME)/buildroot/output/build/linux-4.4.2/ \
	     M=$(PWD) modules
```

Mise en pratique

1. Module noyau sous GNU/Linux
2. Portabilité du module noyau qui n'accède pas au matériel
3. Création dynamique du point d'entrée dans `/dev`

- Au lieu d'exécuter le résultat de la compilation, nous le lions au noyau :

$$\text{Code .c} \underbrace{\longrightarrow}_{\texttt{Makefile}} \text{module .ko}$$

- `[sudo] insmod` module.ko
- `rmmod` module
- `lsmod`