Exemple de système temps-réel: Real-time Linux

ENSPS 3A option ISAV
ENSPS 3A option GLSR
MATSER ISTI parcours AR
FIP 2A

J. GANGLOFF

Plan du cours

- Introduction :
  - Comparatif des systèmes temps-réel
  - Les systèmes temps-réel "*open-source*"
- Architecture de Real-time Linux
  - Le noyau Linux
  - Le noyau Real-time Linux
  - Le mécanisme des modules noyau
  - Communications *user* <-> RT
  - Caractéristiques typiques
  - Avantages / Inconvénients

Plan du cours

- Fonctions de Real-time Linux
  - Compatibilité POSIX-1003.13
  - Tâche périodique
  - *Handler* d'interruption
  - FIFO / Mémoire partagée
  - Mutex et sémaphores
- Méthodes de développement
  - Exemples
  - Méthodologie de debogage
- Liens

I. Introduction

1.1. Comparatif des systèmes temps-réel: temps-réel "mou"

- Le temps-réel "mou" (*soft real-time*) : résolution remporelle faible (> 10 ms).
  - Norme POSIX-1003.1b : extensions TR pour UNIX.
  - Limitations :
    - liées à l'échantillonnage temporel du noyau (*slicing*).
    - liées à la stratégie d'ordonnancement des tâches.
    - liées au traitement des interruptions matérielles.
  - Exemple : Linux.
    - Conforme à la norme POSIX-1003.1b.
    - Noyau échantillonné à 100 Hz.
    - Dans le meilleur des cas (charge du système faible, pas d'accès aux ressources), la résolution temporelle est de 10 ms.

I. Introduction

1.1. Comparatif des systèmes temps-réel: temps-réel "dur"

- Le temps-réel "dur" (*hard real-time*) : résolution temporelle élevée (< 100 μs).
  - Système monolithique
    - VxWorks : architecture hôte/cible. L'environnement de développement et de supervision (hôte) est sous UNIX. Il communique avec le système temps-réel (cible) grâce à une liaison TCP/IP.
  - Système conventionnel modifié
    - RTLinux, RTAI: modification de la couche d'abstraction matérielle pour ajouter des fonctionnalités temps-réelles.
    - Windows 2000 + INTime (TenASys, www.tenasys.com): même concept que RTLinux appliqué à Windows 2000.

I. Introduction

1.1. Comparatif des systèmes temps-réel: temps-réel "dur"

- Système à base de micro-noyau : QNX.
  - UNIX compatible POSIX-1003.
  - Noyau de taille minimale (< 10 Ko) -> temps de réponse minimum.
  - Le noyau implémente seulement 4 services :
    - ordonnancement des processus
    - communications inter-processus (IPC)
    - gestion bas-niveau du réseau
    - dispatching des interruptions
  - Les gestionnaires de ressources fonctionnent comme les autres processus et sont préemptibles par les tâches temps-réel.
  - Point faible : performance (fréquence élevée de changement de contexte, débit important des IPC)

I. Introduction

1.2. Les systèmes temps-réel "*open-source*"

- Les 2 principaux systèmes temps-réel "*open-source*" sont :
  - RTAI (www.rtai.org)
  - RTLinux (www.fsmlab.com)
- Basés sur un noyau linux standard modifié.
- Architecture similaire, API compatible. RTAI possède des fonctionnalités supplémentaires. Licences légèrement différentes.
- Conformes à la norme POSIX-1003.13
- Modification de la couche d'abstraction matérielle pour ajouter des fonctionnalités temps-réel à linux. Cette architecture est basée sur un micro-noyau dont la tâche de plus basse priorité est le système linux.

II. Architecture de Real-time Linux

2.1. Le noyau Linux

- Linux :
  - Linux est un unix-like (AIX, HP-UX, Solaris, ...)
  - Développement initié par Linus Torvalds en 1991
  - Le code source est ouvert et est placé sous la GNU General Public License -> LINUX EST GRATUIT
  - Plateformes supportées : I386, Alpha, Sparc, 680x0, PowerPC, IBM S/390, ... plus de 15 en janvier 2003.
- Linux *vs* les UNIX commerciaux :
  - Architecture monolithique : comme les autres UNIX (excepté Mach 3.0 de Carnegie Mellon).
  - *Multithreading* : comme les autres UNIX

II. Architecture de Real-time Linux

2.1. Le noyau Linux

- Noyau non préemptif : seul Solaris est un noyau pleinement préemptif
- Support multiprocesseur symétrique (SMP) : comme la plupart des UNIX
- Système de fichiers journalisé : comme la plupart des UNIX

- Avantages de Linux :
  - Gratuité
  - Totalement personnalisable : accès aux sources
  - Plateforme matérielle peu onéreuse
  - Performances : un des UNIX les plus rapides
  - Qualité : extrême stabilité

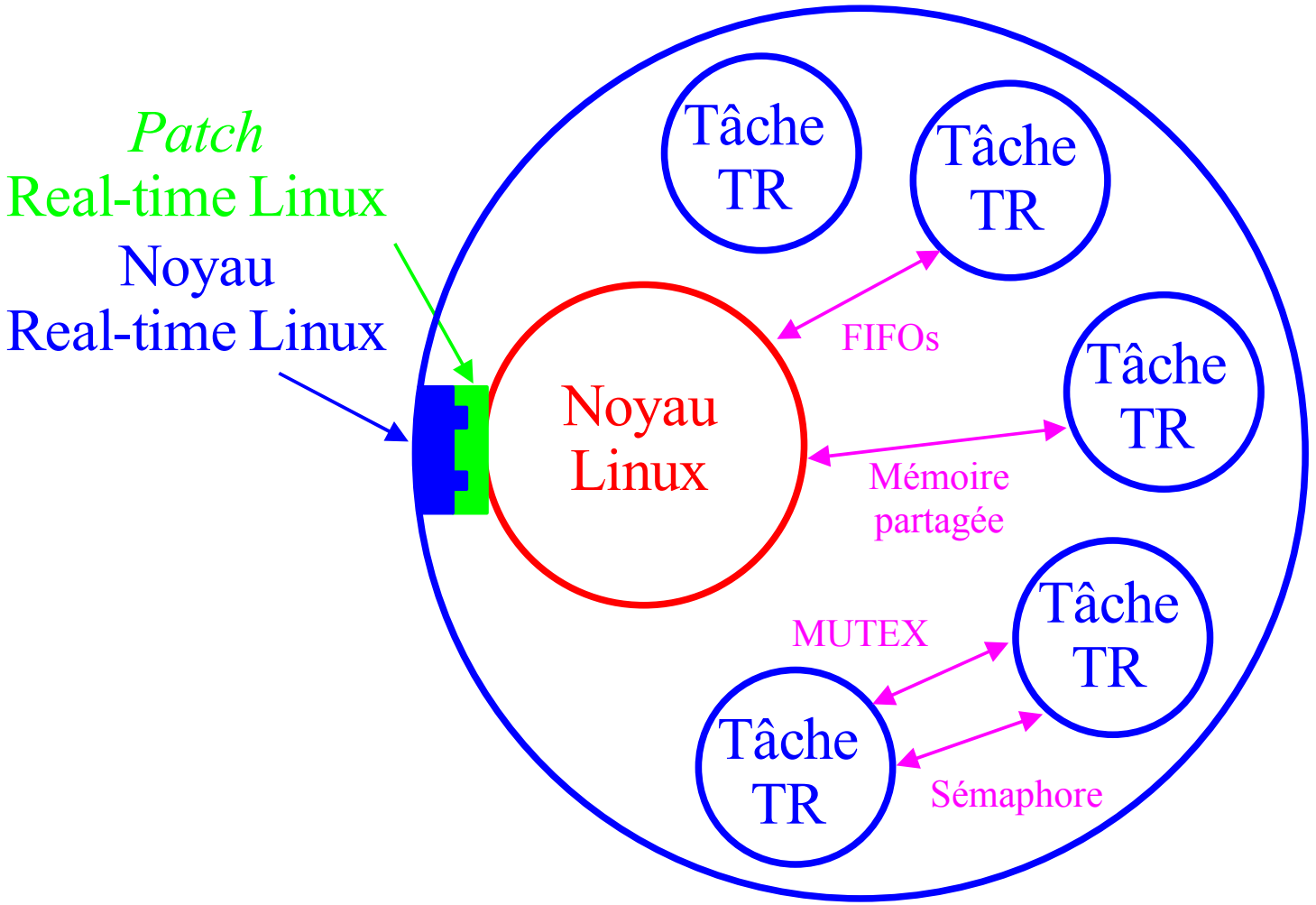
II. Architecture de Real-time Linux

2.1. Le noyau Linux

- Compacité : 2.5 millions de lignes de code contre 30 millions pour windows 2000. Le noyau tient sur une disquette de 1.4 Mb.
- Très bonne compatibilité : reconnaît tous les systèmes de fichiers existants. Compatible avec tous les protocoles réseau. Possibilité d'exécuter des applications compilées pour d'autres systèmes d'exploitation sous Linux.
- Support technique très efficace : groupes de discussion, listes de diffusion, ...
- Modularité : possibilité d'insertion automatique de gestionnaires de périphérique par un mécanisme de modules noyau : fonctionnalité utilisée par Real-time Linux.

II. Architecture de Real-time Linux

2.2. Le noyau Real-time Linux



II. Architecture de Real-time Linux

2.3. Le mécanisme des modules noyau

Module

```
int init_module( void ) {

    /*
        Constructeur
    */

    }
```

```
void cleanup_module( void ) {

    /*
        Destructeur
    */

    }
```

Fonction 1	Fonction 1	- - - -	Fonction n

II. Architecture de Real-time Linux

2.3. Le mécanisme des modules noyau

- Gestion des modules :
  - `insmod <nom du module>` : insertion du module dans le noyau.
    - Vérification de la compatibilité de version
    - Edition de lien dynamique avec les fonctions du noyau.
    - Execution de `init_module`.
  - `rmmod <nom du module>` : retrait du module
    - Vérification que le module n'est pas utilisé.
    - Execution de `cleanup_module`.
  - `lsmod` : liste de tous les modules contenus dans le noyau.

II. Architecture de Real-time Linux

2.3. Le mécanisme des modules noyau

- Les modules RTLinux :
  - `rtl_sched.o` : *scheduler* temps-réel. API POSIX.
  - `rtl_fifo.o` : API de gestion des FIFOs temps-réel.
  - `rtl_posixio.o` : API d'opérations bas-niveau sur les *devices*.
  - `rtl_time.o` : API de gestion du temps (précision : nanoseconde)
  - `rtl.o` : micro-noyau temps-réel.
- L'insertion du module `rtl.o` lance le noyau temps-réel.
- Les modules RTLinux doivent être insérés préalablement à l'insertion de tout module définissant des tâches temps-réel.

II. Architecture de Real-time Linux

2.3. Le mécanisme des modules noyau

- Les modules RTAI :
  - `rtai.o`: API de base de RTAI, dispatcheur d'interruptions, gestion du temps.
  - `rtai_sched.o` : *scheduler* temps-réel
  - `rtai_fifos.o` : fifos et sémaphore

II. Architecture de Real-time Linux

2.4. Communications *user* <-> RT

- Définitions :
  - Processus *user* : processus tournant en mode *user*, càd tout processus Linux géré par le noyau Linux. Les processus *user* bénéficient des mécanismes de protection de l'accès aux ressources: ils ne peuvent pas compromettre l'intégrité du système.
  - Processus RT : processus tournant en mode *Kernel*. Toutes les tâches Real-time Linux sont lancées à l'insertion d'un module et tournent donc en mode *Kernel*. Elles ne bénéficient pas des mécanismes de protection des processus *user* et peuvent donc compromettre l'intégrité du système. Une programmation minutieuse est donc nécessaire.

II. Architecture de Real-time Linux

2.4. Communications *user* <-> RT

- Il est nécessaire de prévoir des dispositifs de communication entre Linux et RTLinux.
  - FIFOs : il existe 64 FIFOs temps-réel (`/dev/rtf0` à `/dev/rtf63`) qui permettent une communication bi-directionnelle entre Linux et Real-time Linux. Ces FIFOs peuvent éventuellement servir de moyen de communication entre 2 tâches temps-réel.
  - Mémoire partagée : utilisation d'un *device* `/dev/mbuff` pour partager une zone mémoire entre un processus RT et un processus *user*.
    - Avantage : pas de transfert de données : gain de temps
    - Inconvénient : prévoir un mécanisme de mutex
  - *Buffer* circulaire du noyau : affichage de messages de débogage. Consultable avec la commande `dmesg`.

II. Architecture de Real-time Linux

2.4. Communications *user* <-> RT : exemple

- API 100% compatible entre RTAI et RTLinux
- Fonctions de base :
  - `int rtf_create(unsigned int fifo, int size);` création d'une fifo.
  - `int rtf_create_handler(unsigned int fifo, int (*handler)(unsigned int fifo));` création d'un *handler* de fifo
  - `int rtf_destroy(unsigned int fifo);` destruction d'une fifo
- Programme de test

II. Architecture de Real-time Linux

2.5. Caractéristiques typiques

- Caractéristiques typiques de Real-time Linux :
  - Fonctionne sur architecture i386, mono ou multi-processeurs, avec linux version 2.2 ou 2.4.
  - Taille du *patch* du noyau linux (source) : < 100 Ko
  - Taille des modules Real-time Linux (objet) : < 100 Ko
  - *Jitter* maximum sur un tâche périodique (pentium III 800) : 2 μs.
  - Temps de réponse aux interruptions matérielle : < 2 μs.
  - Résolution du *timer* : 1 ns
  - *Scheduler* basé sur la priorité : la tâche de priorité la plus haute est executée.

II. Architecture de Real-time Linux

2.6. Avantages / inconvénients

- Avantages :
  - On bénéficie de tout l'environnement UNIX classique (outils de développement, Xwindows, réseau, ...) tout en ayant des fonctionnalités temps-réel. Les 2 pouvant communiquer *via* les FIFOs ou la mémoire partagée.
  - Le micro-noyau Real-time Linux garantie des temps de commutation de contexte très courts -> bonnes performances.
- Inconvénients :
  - Programmation en mode *kernel* -> pas de mécanisme de protection de l'intégrité du système
  - API limitée et très simple comparée à des systèmes temps-réel commerciaux.

III. Fonctions de Real-time Linux

3.1. Compatibilité POSIX-1003.13

- POSIX 1003.13 : "*minimal realtime operating system*"
  - Fonctions portables : `pthread_create`, `pthread_join`, `pthread_kill`, `pthread_cancel`, `sigaction`, fonctions de gestion des mutex, fonctions de gestion des sémaphores, ...
  - Fonctions spécifiques : `pthread_make_periodic_np`, `pthread_setfp_np`, `pthread_suspend_np`, `pthread_wait_np`, `pthread_wakeup_np`, `pthread_attr_setcpu_np`.
- En n'utilisant que des fonctions portables compatibles POSIX, il est possible d'exécuter le programme en mode *user* avec des contraintes temps-réel "mou", et ainsi de le déboguer plus facilement avant de le re-compiler pour le mode TR "dur".

III. Fonctions de Real-time Linux

3.2. Tâche périodique

```
pthread_t               mytask;
int init_module ( void ) {
    struct sched_param   p;
    pthread_attr_t       attr;
    hrtime_t             now = gethrtime();

    pthread_attr_init( &attr );
    pthread_attr_setfp_np( &attr, 1 );

    pthread_create ( &mytask, &attr, fun, (void *) 1 );
    pthread_make_periodic_np ( mytask, now + 2 * NSECS_PER_SEC, 31230000 );
    p . sched_priority = 1;
    pthread_setschedparam ( mytask, SCHED_FIFO, &p );
    return 0;
    }

void cleanup_module ( void ) {
    pthread_delete_np ( mytask );
    }

void *fun ( void *t ){
    while( 1 )   {
        pthread_wait_np( );
        /* Code executé périodiquement */
        }
    }
```

Fonction retournant le temps courant (ns)

Initialisation de la structure définissant la tâche

Autorisation des calculs en virgule flottante

Création de la tâche

Période (ns)

Instant de départ

Priorité de la tâche

Destruction de la tâche à l'extraction du module

Attente du prochain top du *timer*

III. Fonctions de Real-time Linux

3.3. *Handler* d'interruption

```
int init_module ( void ) {

    rtl_request_irq( 8, int_handler );

    rtl_hard_enable_irq( 8 );

    return 0;
    }

void cleanup_module ( void ) {

    rtl_free_irq( 8 );
    }

unsigned int int_handler( unsigned int irq, struct pt_regs *regs ) {

    pthread_wakeup_np ( thread );

    /* clear IRQ */

    rtl_hard_enable_irq( 8 );

    return 0;
    }
```

numéro de l'interruption

pointeur sur le *handler* d'interruption

activation de l'interruption 8

libération du *handler* l'interruption 8

réveille de la tâche décrite par `thread`

réactivation de l'interruption 8
(le *handler* est appelé avec les interruptions désactivées)

III. Fonctions de Real-time Linux

3.4. FIFO / Mémoire partagé

FIFOs : voir exemple 1

Mémoire partagée :

Le contenu de la mémoire partagée peut changer à tout moment -> volatile

Pointeur identique

```
volatile char *shm;

int init_module ( void ) {

    shm = (volatile char*)
        mbuff_alloc( "demo",1024*1024 );

    if( shm == NULL ) {
        rtl_printf( "alloc failed\n" );
        return -1;
        }

    sprintf( (char*)shm, "data\n" );
    return 0;
    }

void cleanup_module ( void ) {

    if ( shm )
        mbuff_free( "demo",(void*)shm );
    }
```

```
#include <stdio.h>

#include "mbuff.h"

volatile char *shm;

main ( int argc,char *argv[] ){

    shm = (volatile char*)
    mbuff_alloc( "demo", 1024*1024 );

    if( shm == NULL ) {
        printf( "alloc failed\n" );
        exit( 2 );
        }

    printf( "mbuff: %s\n", shm );

    mbuff_free( "demo", (void*)shm );
    return( 0 );
    }
```

Identification de la zone partagée

Affichage en mode *user* de la chaîne "data" définie en mode *kernel*

III. Fonctions de Real-time Linux

3.5. Mutex et sémaphores

```
static pthread_mutex_t   mutex;
static pthread_t         threads[2];

int init_module ( void ) {
    pthread_attr_t attr;

    pthread_mutex_init( &mutex, 0 );
    pthread_attr_init ( &attr );
```

Création d'une *thread* sur le CPU0

```
    pthread_attr_setcpu_np( &attr,0 );
    pthread_create ( &threads[0],
                     &attr, start0,
                     (void *) 0 );
    pthread_attr_setcpu_np( &attr,1 );
    pthread_create ( &threads[1],
                     &attr, start1,
                     (void *) 0 );

    return 0;
    }
```

Création d'une *thread* sur le CPU1

```
void cleanup_module ( void ) {
```

Attente que les 2 *thread* soient finies

```
    pthread_join( threads[0], NULL );
    pthread_join( threads[1], NULL );
    pthread_mutex_destroy( &mutex );
    }
```

```
static void * start0( void *arg ) {

    pthread_mutex_lock ( &mutex );
    outb( 0xff, 0x378 );
    nanosleep( hrt2ts(10000), NULL );
    outb( 0x00, 0x378 );
    nanosleep( hrt2ts(10000), NULL );
    outb( 0xff, 0x378 );
    pthread_mutex_unlock ( &mutex );

    return (void*) 0;
    }
```

Pulse de 10 μs sur le port parallèle

```
static void * start1( void *arg ) {

    pthread_mutex_lock ( &mutex );
    outb( 0xff, 0x378 );
    nanosleep( hrt2ts(10000), NULL );
    outb( 0x00, 0x378 );
    nanosleep( hrt2ts(10000), NULL );
    outb( 0xff, 0x378 );
    pthread_mutex_unlock ( &mutex );

    return (void*) 1;
    }
```

III. Fonctions de Real-time Linux

3.5. Mutex et sémaphores

- Les mêmes fonctionnallités peuvent être obtenues avec les sémaphores :
  - `sem_wait, sem_timedwait` : permet de verrouiller un sémaphore avec ou sans *timeout*.
  - `sem_post` : permet de déverrouiller un sémaphore.
- Les mutex et les sémaphores sont entièrement compatibles avec la norme POSIX-1003.13 et peuvent donc être implémentés en mode *user* comme en mode *kernel*.

III. Fonctions de Real-time Linux

3.6. LXRT

- LXRT est une fonctionnalité propre à RTAI qui n'a pas d'équivalent sous RTLinux :
- Permet d'exécuter un processus temps-réel en mode *user*
- Avantages :
  - Permet de développer de manière plus sûre en bénéficiant du mécanisme de protection de la mémoire
  - Utilisation des outils de debogage classiques
  - Pas besoin d'être *root* pour lancer le programme temps-réel
  - API identique au mode kernel
  - Portage des applications plus simple
- Inconvénient :
  - Temps de commutation de contexte doublé

IV. Méthodes de développement

4.1. Exemples sous RTAI

- "stress" : permet de mesurer en nanosecondes le *jitter* de l'instant de début d'une tâche périodique cadencée à 1000 Hz répétée 500 fois.
- "sound" : permet, par une modulation de largeur d'impulsion d'un signal carré à 8000 Hz, de produire un son sur le haut-parleur standard d'un PC. Seul un système TR peut garantir un son sans "crachottements".

IV. Méthodes de développement

4.2. Méthodologie de débogage

- Deux symptômes fréquents :
  - Blocage du système : le processus temps-réel prend toute la ressource processeur. Il n'y a plus de temps pour executer linux. Solution : rechercher le *bug* dans les boucles (`for`, `while`) et plus précisément dans leur condition de sortie.
  - Message *kernel* dans `dmesg` : "Kernel page fault", "oops", ... Dans ce cas, l'erreur provient d'un dépassement de tableau. Il est nécessaire de rebooter le système sous peine de compromettre son intégrité (corruption de *filesystem*).
- Pour "traquer" un *bug* : utiliser `rtl_prinf` (RTLinux) ou `printk` (RTAI) pour visualiser l'état de variables importantes du programme. Trop de messages peuvent bloquer le système.

V. Liens

- www.realtimelinux.org : portail d'accès à toutes les pages ayant rapport avec le temps-réel sous linux.
- www.fsmlab.com : site web officiel de RTLinux.
- www.rtai.org : site web officiel de RTAI.
- www.linuxhq.com : site web sur le développement de linux.
- www.kernel.org : accès aux sources de linux.