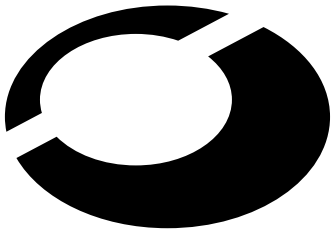
www.alcove.fr

RTAI

Stéphane List
Stephane.List@fr.alcove.com

version 1.18



l'informatique est libre

Alcôve

Alcôve
Centre Paris Pleyel
153 bd Anatole France
93200 Saint-Denis, France

Tél. : +33 1 49 22 68 00
Fax : +33 1 49 22 68 01
E-mail : alcove@alcove.fr, Toile : www.alcove.fr

Table des matières

Chapitre 1

Définition et concepts de RTAI

Les temps réel...

Définition du IEEE : *"Un système temps réel est un système dont le temps de réponse est aussi important que la qualité de fonctionnement."*

Déterminisme != vitesse : *"Le temps réel est la capacité à répondre à une sollicitation en un temps déterminé pour produire une réaction appropriée."*

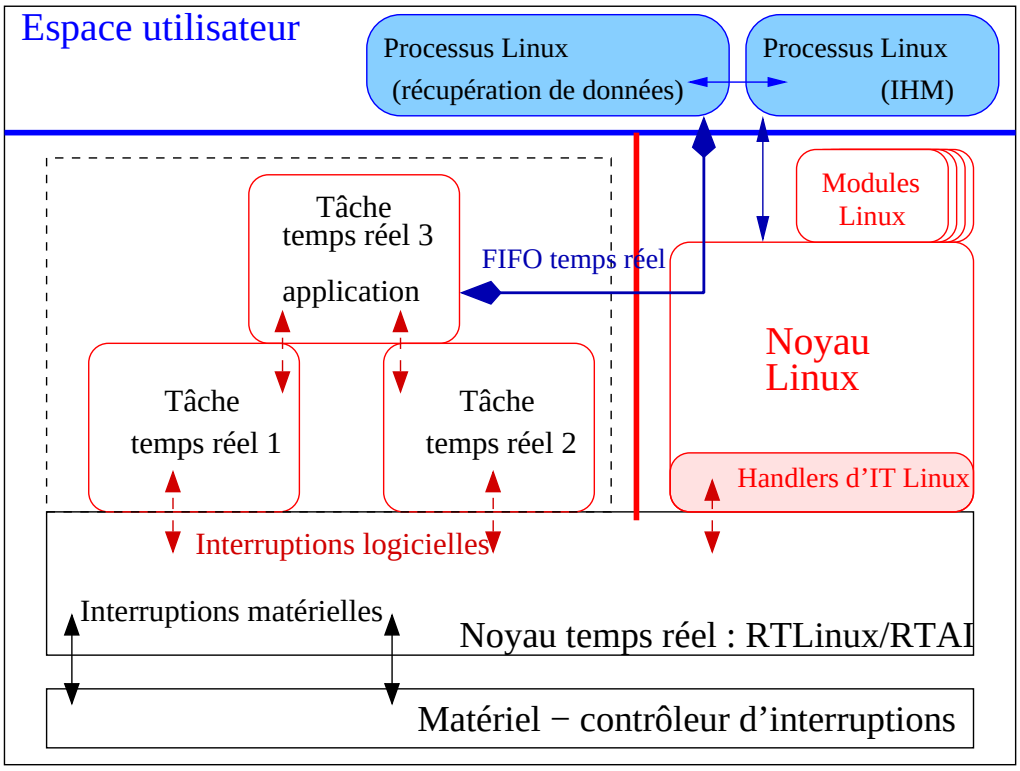
Multitude de définitions du temps réel qui peut être qualifié de mou/dur, critique...

guaranteed worst case vs best effort

Architecture micro-noyau

- Les tâches
  - Linux est la tâche temps réel de plus faible priorité
  - Dialogue entre tâches temps réel et processus Linux par FIFOs ou mémoire partagée
- Les interruptions
  - Le code de masquage et démasquage des interruptions a été réécrit
  - Les interruptions « hard » sont déroutées par la couche temps réel
  - Générations d’interruptions « soft » à Linux
  - Linux ne peut intervenir sur les interruptions « hard »

Architecture



Définition et concepts de RTAI

Alcôve - RTAI

Implémentation

- Les tâches RTAI se présentent sous forme de modules noyau (chargeables dynamiquement)
- Allocation statique de toutes les ressources
- Communications non-bloquantes côté temps réel

Outils

Par exemple, RTAI est structuré comme partie centrale minimale sur laquelle vient se charger une collection de modules qui fournissent des services optionnels ou des niveaux d'abstraction.

- `rtai_sched` : scheduler
- `rtai_shm` : Shared Memory
- `rtai_fifos` : FIFOs
- `rtai_watchdog` : Watchdog
- `rtai_pqueue / pthread / sched_mem / utils` : API POSIX
- ...

=>Système modulaire et extensible.


Définition et concepts de RTAI
Alcôve - RTAI

Besoin d'un nouveau modèle de programmation

Découpage de l'application :

"Si un service est intrinsèquement non temps réel, il doit être fourni par Linux et non par les modules RTAI."

Gestion de la mémoire : allocation statique / pas de protections...

Communications non-bloquantes côté temps réel.

Utilisation de modules chargeables / API spécifique.

Installation de RTAI

- Télécharger RTAI
- Chercher les versions de Linux supportées par RTAI
- Télécharger Linux
- Patcher Linux avec RTAI
- Configurer / Compiler / Installer le noyau
- Rebooter

Installation de RTAI avec Linux Trace Tollkit (LTT)

- Télécharger Linux 2.4.0-test10
- Télécharger RTAI 24.1.2
- Télécharger LTT 0.9.4
- Patcher RTAI avec LTT
- Patcher Linux avec RTAI-LTT
- Configurer / Compiler / Installer le noyau
- Rebooter

Utilisation de RTAI avec LTT

- insmod rtai_trace
- insmod tracer
- insmod rtai
- modules nécessaires : [insmod rtai_fifos rtai_shm rtai_sched]
- insmod mon_module
- tracedaemon -ts10 /dev/tracer ./out.trace ./out.proc
- tracevisualizer

Critères de choix

- Avantages :
  - Comportement temps réel critique
  - Modularité, scalabilité
  - Compacité du code et du patch noyau (64 ko/qq 100 lignes)
    - Version récente du noyau Linux (dernières innovations)
    - Peu de bugs, correction rapide, testé par de nombreux spécialistes
- Inconvénients :
  - Modification du code des drivers pour un comportement temps réel
  - Apprentissage d'une API (facilité par le module POSIX)
  - Difficulté de débogage (espace noyau).

L’ordonnanceur RTAI

Ordonnanceurs

RTAI contient 3 schedulers différents, le scheduler UP est installé par défaut.

– UP : Uni-Processor
– SMP : Symetrical Multi-Processor
– MUP : Multi Uni-Processor

Remarque : Attention lors de la compilation du noyau Linux de ne sélectionner SMP que si la machine est bien SMP.

Choix de l'ordonnanceur

- rt_set_oneshot_mode()
- rt_set_periodic_mode()

Ordonnanceur : Mode One shot

- rt_set_oneshot_mode()
- paramètre de start_rt_timer ignoré
- calibration du jitter avec */latency_calibration*
- Avant Pentium : utilisation du rdtsc (Read Time Stamp Clock), registre counter2 du chip 8254
- Depuis Pentium : utilisation du CPU TSC (Time Stamp Clock)

Ordonnanceur : Mode périodique

- rt_set_periodic_mode()
- mode par défaut
- calibrage inutile
- les tâches dont la période n’est pas un multiple de la période du timer sont servies au mieux.
- Utilise le chip 8254
- *less flexible but more efficient*

Politiques d’ordonnancement

- RT_SCHED_FIFO : politique par défaut
- RT_SCHED_RR : Round Robin, un quantum de temps alloué aux taches
- RMS : Rate Monotonic scheduling . La tache périodique de fréquence la plus élevée est la plus prioritaire.

voir rtai/README.SCHED_POLICY

Priorité des tâches

- 0 : RT_HIGHEST_PRIORITY = WATCHDOG_PRIORITY
- 0x3fffFfff : RT_LOWEST_PRIORITY
- 0x7fffFfff : RT_LINUX_PRIORITY

Déclaration d’une tâche

```
#include "rtai_sched.h"
rt_task_init (RT_TASK *task, // Tâche
void (*rt_thread)(int), // Code du thread
int data, //paramêtre d’appel du thread
int stack_size, //taille de la pile
int priority, //priorité de la tâche
int uses_fpu, //0 = sans FPU, 1 = avec FPU
void (*signal)(void)); // fonction appelée dans le
contexte de la tâche, chaque fois que celle-ci
devient active
```

Programmation des tâches périodiques

- rt_task_init : initialise une tâche
- rt_task_make_periodic : rend une tache périodique
- rt_task_wait_period : rend la main jusqu’à la période suivante
- rt_task_delete : supprime une tâche

Programmation des tâches apériodiques et/ou sporadiques

- rt_task_init : initialise une tâche
- rt_task_resume : reprend l’execution d’une tâche
- rt_task_yield : rend la main au scheduler (yield != suspend)
- rt_task_delete : supprime une tâche

Gestion du timer

- start_rt_timer() : démarre le timer
- rt_get_time() : donne l’heure
- rt_get_cpu_time_ns() : donne l’heure precisemment en mode périodique
- rt_sleep() : endort la tâche pendant une durée (deschedule)
- rt_busy_sleep() : attend une durée (ne deschedule pas, attente active dans une boucle while)


L’ordonnanceur RTAI
Alcôve - RTAI

Gestion globale des Interruptions

- rt_global_save_flag_and_cli ou rt_global_cli
- < section critique >
- rt_global_restore_flags
- Contrairement à Linux, RTAI n’offre pas d’interface de type *spinlock* .

Installer un handler d’interruptions

- rt_request_global_irq : installe un handler pour une IRQ
- rt_pend_linux_irq : invoque le handler de Linux pour cette IRQ
- rt_free_global_irq : désinstalle un handler d’IRQ
- rt_request_linux_irq : installe un handler qui partage l’IRQ avec Linux
- rt_free_linux_irq : désinstalle le handler

Fonctions utiles de RTAI

- rt_printk
- rtai_print_to_screen

Watchdog RTAI

- Watchdog = tâche de plus haute priorité
- Lors du codage, vérifier que aucune tâche n’arrête le timer
- Pour définir la politique du Watchdog : rd_wdset_policy()
- Les sentences appliquées sont consultables dans /proc/rtai/watchdog

Politiques de Watchdog RTAI

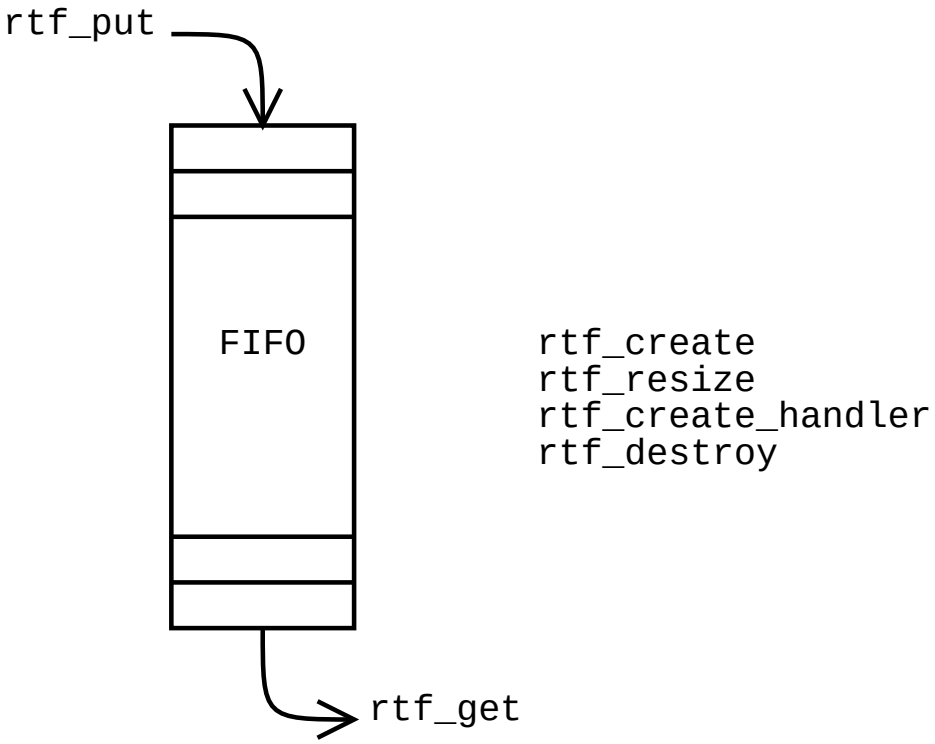
- NOTHING Non recommandée
- RESYNC Pour des dépassements occasionnels
- DEBUG Pour debugger les tâches oneshot
- STRETCH Pour augmenter la période des tâches qui débordent
- SLIP Pour les boucles infinies
- SUSPEND Pour le cas général
- KILL Peine de mort (pas de traces dans /proc)

Paramétrage du Watchdog RTAI

- Grace Period : délai avant application de la sentence
- Offense Limit : Suspension des tâches qui abusent de la sentence SLIP et STRETCH
- Safety Net : Evite les boucles infinies en mode STRETCH

FIFO : Communication entre RTAI et Linux

FIFO : First In First Out



FIFO : Communication entre RTAI et Linux

Alcôve - RTAI

FIFOS

- créées en mode user ou Kernel
- peuvent être nommées et référencées par ce nom
- peuvent être *retaillée*
- entrée dans /proc
- accessibles dans le file-system : /dev/rtfx (x=0..5)

FIFOS

- synchronisation avec sémaphores
- multiples écrivains et lecteurs
- lecture/écriture *timed* (avec timeout, donc non bloquantes)
- signaux asynchrones pour notification d'évènements
- écriture : rtf_put
- lecture par polling : tâche périodique + rtf_get
- lecture par handling : lecture par le handler de la FIFO

FIFOS : gestion avec les sémaphores

- rtf_sem_init
- rtf_sem_wait
- rtf_sem_try_wait
- rtf_sem_timed_wait
- rtf_sem_post
- rtf_sem_destroy

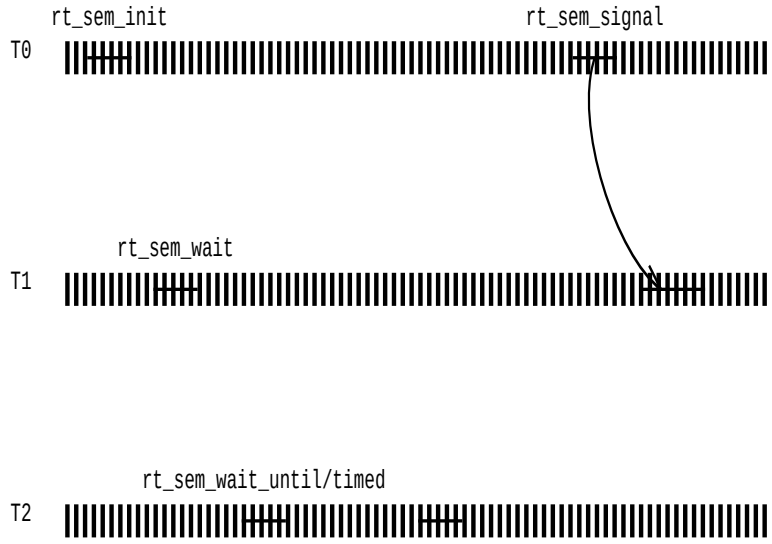
FIFOS : gestion avec plusieurs lecteurs/écrivains

- rtf_read_all_at_once
- rtf_read_timed
- rtf_write_timed
- rtf_suspend_timed

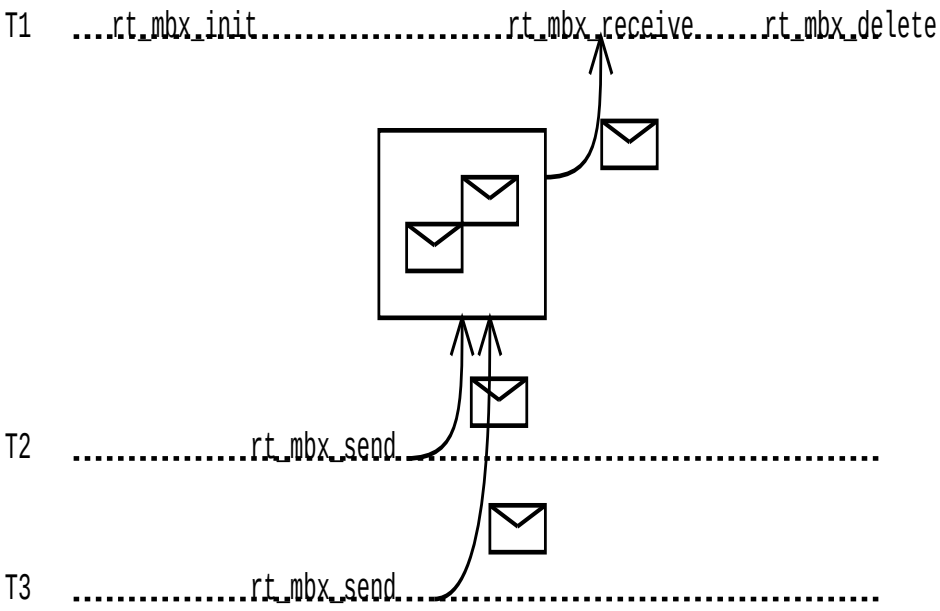
Chapitre 4

Fonctions avancées

Gestion des sémaphores



Gestion des boites aux lettres



Gestion des boites aux lettres

- rt_mbx_send : envoie (bloquant)
- rt_mbx_send_wp : envoie le + possible (tronque si besoin)
- rt_mbx_send_if : envoie si possible (non bloquant)
- rt_mbx_send_until/timed : envoie tant que le message n'est pas OK (avec timeout)

Gestion des RPC : Remote Procedure Call

Similaire aux messages QNX.

- rt_rpc : fait appel à une fonction externe ;
- rt_rpc_if : fait appel à une fonction externe si la tâche est prête a répondre ;
- rt_rpc_until/timed : appel avec time out.

Mémoire partagée : Mbuff / Shmem

– La mémoire partagée est plus complexe à utiliser que les FIFOS. Lorsque 2 applications requièrent un *Hand Shaking* un protocole doit être défini.
– L'exclusion mutuelle entre Linux et les tâches RTAI n'est pas garantie.

Mémoire partagée avec Shmem

– Dans l'espace utilisateur
  • adr = rtai_malloc(name, size) ;
  • adr2 = rtai_malloc_adr(start_adress,name,size) ;
  • rtai_free(name, adr) ;
– Dans l'espace noyau
  • adr = rtai_kmalloc(name, size) ;
  • rtai_kfree(name) ;

Attention aux droits du device /dev/rtai_shm pour accéder aux mémoires partagées sans être root.

Mémoire partagée avec Mbuff

- mknod /dev/rtai_shm c 10 254
- Informations dans /proc/mbuff
- Dans l'espace utilisateur
  - adr = mbuff_alloc(name, size) ;
  - adr2 = mbuff_alloc_at(name,size,start_adress) ;
  - mbuff_free(name, adr) ;
- Dans l'espace noyau
  - adr = rtai_kmalloc(name, size) ;
  - rtai_kfree(name) ;


Fonctions avancées
Alcôve - RTAI

IPC : Inter-Process Communications provided by POSIX modules

- Message Queue : mq_open / close, mq_send / receive..
- Pthread : pthread_create, sched_yield..
- Mutex : pthread_mutex_init, pthread_mutex_lock
- Variable conditionnelle : pthread_cond_init

Limations du module POSIX

- Les pthread sont executés dans le contexte d'un unique process. Donc il n'y a pas de relation pere/fils, et les fonctions pthread_join et pthread_detach n'execute rien.
- Pas de signal handling
- Absolute time non supporté : pas de fonction pthread_cond_timedwait
- Les message queue ne sont pas dirigées vers les FIFO, il n'y a donc aucune communication avec l'espace User.
- Les sémaphores POSIX ne sont pas implémentées (shm_open...)

LXRT (LinuX Real Time)

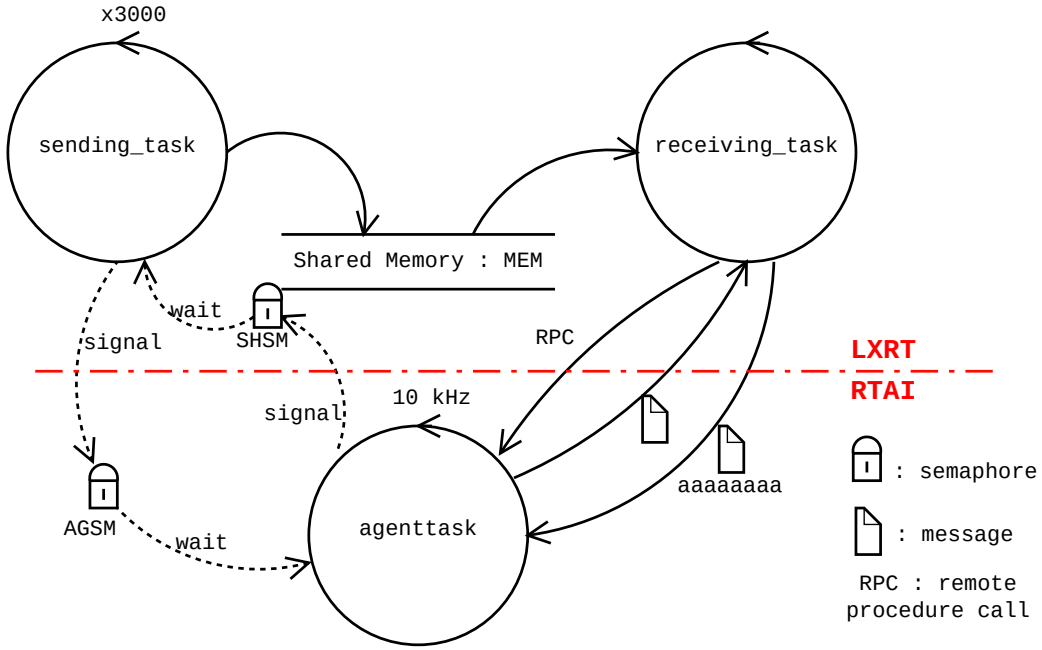
Fonctionnalités de LXRT

- LXRT permet d’exécuter des tâches temps-réel en mode user
- Ces taches s’executent avec les méchanismes de protection mémoire de Linux
- Les taches de LXRT possèdent une plus grande priorité et une meilleure granularité de scheduling que les taches Linux.
- Les tâches peuvent être debuggées en mode linux puis transférées dans l’espace kernel
- Les utilisateurs NON root peuvent créer des tâches temps réel.

Performances de RTAI / LXRT

- Sur un celeron 534 MHz
- Jitter moyen de RTAI : 2us
- Jitter max de RTAI : 18us
-
- Jitter moyen de LXRT : 5us
- Jitter max de LXRT : 21us

Illustration de rtai/lxrt/rt_agent



Licence

RTAI

Le code de RTAI est régi par la LGPL (Lesser General Public License). Cette déclinaison de la licence GPL permet, la non redistribution du code source des applications construites à partir de celle-ci. Cependant, toute modification sur du code LGPL doit être rendue publique en cas de diffusion.

RTLinux

Le code de RTLinux est sous GPL (General Public License) + US patent. Le brevet n’est valable qu’aux US pour l’instant, et stipule que l’on ne peut vendre un produit utilisant RTLinux que s’il est GPL, et que fsmlabs est averti de la transaction. Le 14 Septembre 2001, la FSF a annoncé que la licence de rtlinux viole la licence GPL, et que quiconque redistribue RTLinux, viole aussi la GPL.

Travaux Pratiques

Exercice 1

Ecrire un module RTAI qui écrit *Hello World* dans le fichier log.

Exercice 2

Ecrire un module RTAI avec une tâche NON périodique qui écrit *Hello World* 10 fois dans le fichier log.

Exercice 3

Ecrire un module RTAI avec une tâche PERIODIQUE (T=0.1 ms) qui écrit *Hello World* 10 fois dans le fichier log.

Utiliser les fonctions nano2count et count2nano pour faire les conversions entre les unités du timer et le temps en nanosecondes.

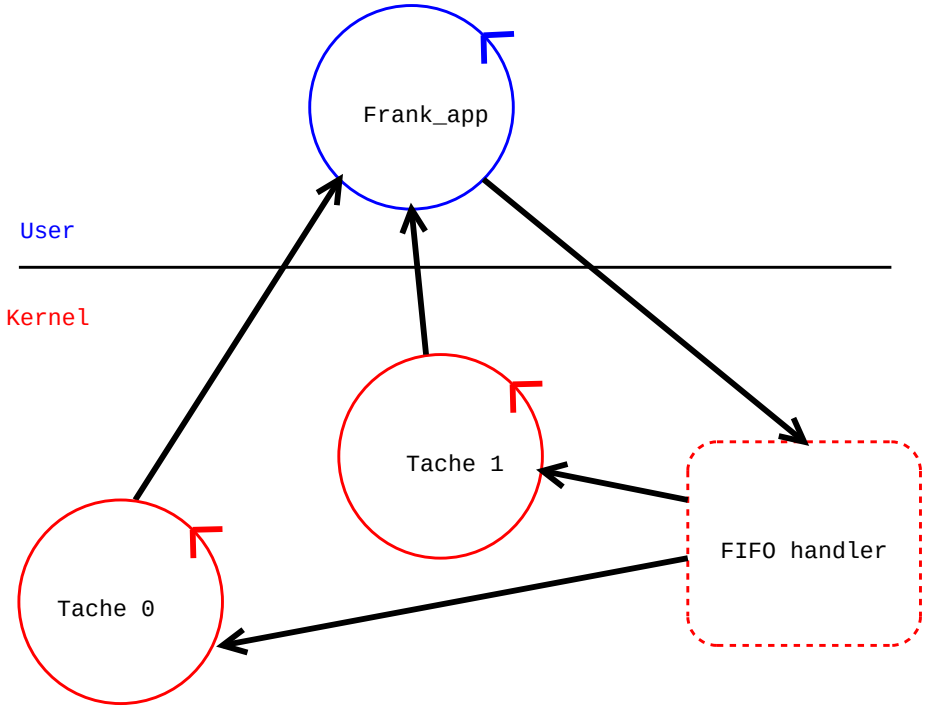
Exercice 4

Ecrire un module RTAI avec une tâche périodique qui écrit un compteur de 1 à 10 dans une FIFO.

Et écrire un programme en mode User qui va lire la FIFO.

Que se passe-t-il si on appelle 2 fois la fonction de création de FIFO pour la même FIFO ?

Exercice 5



TP : Retrouvez les numéros des FIFOS

Exercice 6

Ecrire un module RTAI qui écrit une valeur dans une mémoire partagée. (En utilisant le module rtai_shm)

Et écrire un programme en mode User qui va lire la mémoire partagée.

Exercice 7

Ecrire un module RTAI qui compte les interruptions produites par le clavier et qui écrit la valeur du compteur dans une mémoire partagée.

Et écrire un programme en mode utilisateur qui va lire la mémoire partagée.

(Ecrire le module en utilisant *rt_request_linux_irq* puis avec *rt_request_global_irq*)

Exercice 8

Dans le répertoire latency_calibration.

Mesurer le jitter en mode ONE SHOT.

Mesurer le jitter avec l'ordonnanceur en mode périodique.

Commenter les résultats.

Références

- Portail communautaire RTLinux http ://www.rtlinux.org
- RTLinux http ://www.rtlinux.com
- FSMLabs http ://www.fsmlabs.com
- RTAI http ://www.rtai.org
- Portail communautaire général http ://www.realtimelinux.org
- Portail plus jeune http ://www.realtimelinuxfoundation.org
- Portail Linux embarqué http ://www.linuxdevices.com
- Linux Trace Toolkit http ://www.opersys.com/LTT