Implantation logicielle d'applications temps réel

Mise en oeuvre sur des éléments de RTAI

R.KOCIK – Département Systèmes Embarqués

PLAN

I. Développement d'une application temps réel
II. Quel Système d'exploitation (OS) utiliser ?
III. Un OS Multi-tâches : Linux
IV. Linux temps réel
V. Développer avec RTAI
VI. Taches RT et ordonnancement avec RTAI
VII. Synchroniser, Communiquer
VIII. RTAI: conclusion
IX. Et les autres RTOS ….

I. DÉVELOPPEMENT D'UNE APPLICATION TEMPS-REEL

Cycle de développement

Spécification
- Définir Tâches
- Définir contraintes temporelles

Analyse
- Choix d'algorithme d'ordonnancement
- Analyse d'ordonnançabilité

Mise en oeuvre
- Tâches
- Ordonnancement

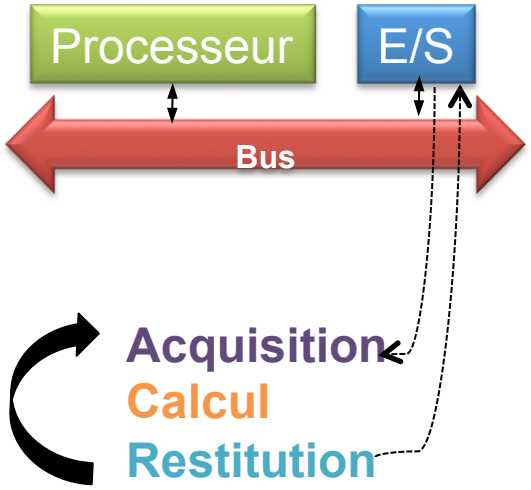
Validation
- Tâches
- Comportement temps réel

Mise en œuvre d'un ordonnancement TR

- Plusieurs solutions possibles
  - Sans exécutif temps réel
  - Avec exécutif temps réel
- Dépend:
  - Caractéristiques des tâches
  - Politique d'ordonnancement (Priorité fixe/dynamique, Préemption/sans préemption)
  - Architecture matérielle
  - Criticité de l'application (fiabilité, sûreté)

Sans exécutif temps réel : principes (1)

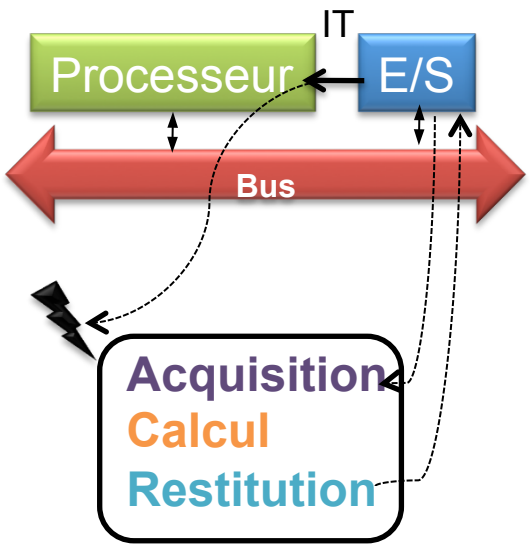
- scrutation




```
faire
    (attendre prochaine échéance)
    faire
        vérifier capteur
    tant que données non disponibles
    lire capteurs
    traiter les données
    faire
        vérifier les actionneurs
    tant que actionneurs non accessibles
    ecrire actionneurs
jusqu 'à arrêt du système
```

Sans exécutif temps réel : principes (2)

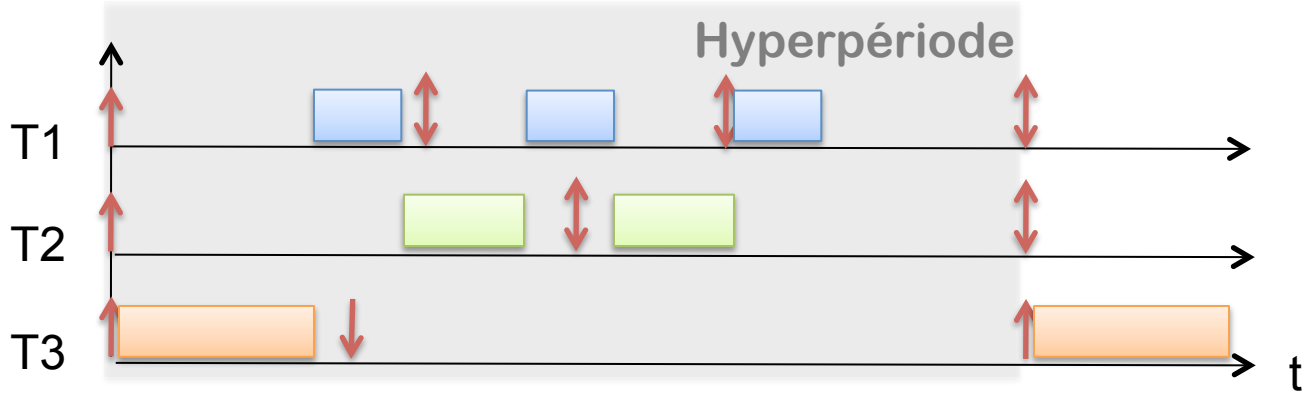
- Interruption (IT ou IRQ)
  - Matérielle (E/S, réseau,…), logicielle (exception), Timer
  - événement qui provoque un changement dans l'exécution normale d 'un programme




```
Sur interruption faire
    lire capteurs
    traiter les données
    faire
        vérifier les actionneurs
    tant que actionneurs non accessibles
    écrire actionneurs
    acquitter interruption
```

Scrutation : méthode (1)

- Analyse : choix et validation d'un ordonnancement
- Construction de l'ordonnancement



- Construction d'une séquence d'exécution sur l'hyperpériode

séquence

Scrutation : méthode (2)

séquence 

- Construction du code

```
Initialisations
```

1 seule exécution : allocation mémoire, Initialisations capteurs et actionneurs …

```
faire
    T3();
    T1();
    T2();
    T1();
    T2();
    T1();
    attente_fin_hyperpériode();
jusqu'à arrêt du système
```

Séquence de tâches
Exécution périodique

```
Finalisations
```

1 seule exécution : désallocation mémoire, sauvegardes, on met le système dans une Configuration sécurisée

Scrutation (3)

- Garantir la périodicité 

1ère sol : En début (ou fin) de séquence on attend la date de début (ou de fin) de l'hyperpériode suivante (ou courante)

```
Fonction attente_debut_hyperpériode()
    Début
    Faire
        Date_courante ← lecture timer
    Tant que date_courante < date_debut_next_HP
    Date_debut_next_HP ← Date_debut_next_HP + Hyperperiode
    fin
```

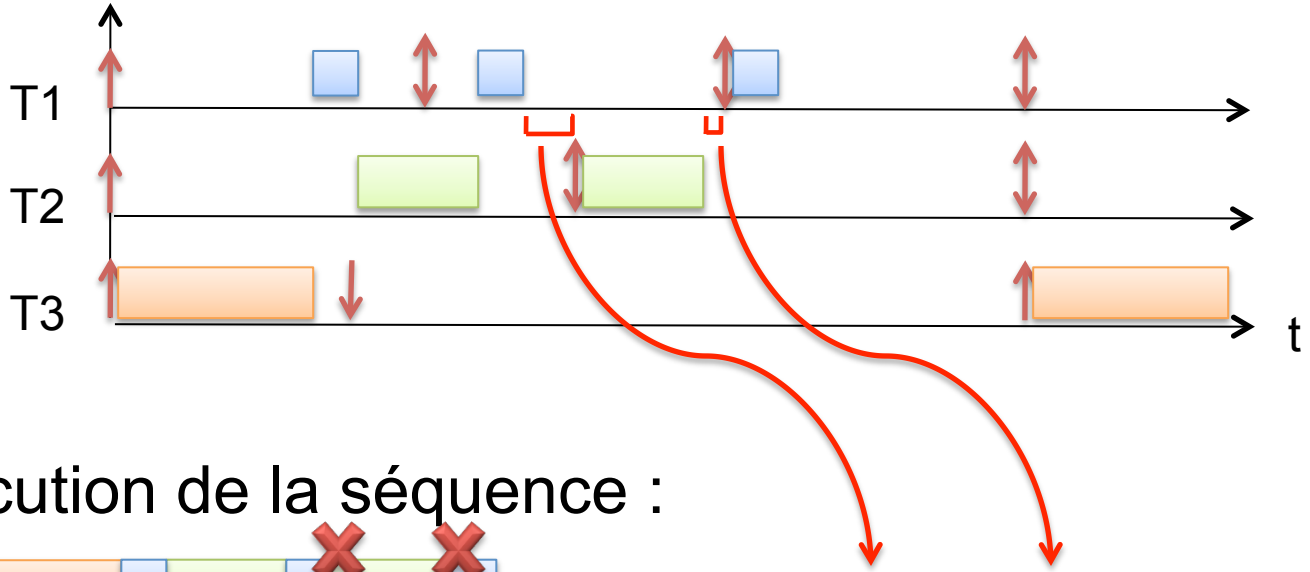
2ème sol : On programme une interruption Timer qui déclenche périodiquement l'exécution de la séquence

- Sinon système autocadencé : 

Dans ce cas Période = Durée d'exécution de la séquence

Scrutation (4)

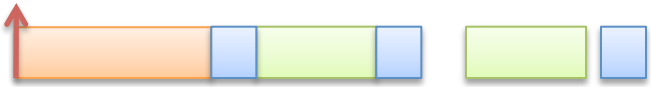
Ordonnancement oisif



- Exécution de la séquence :

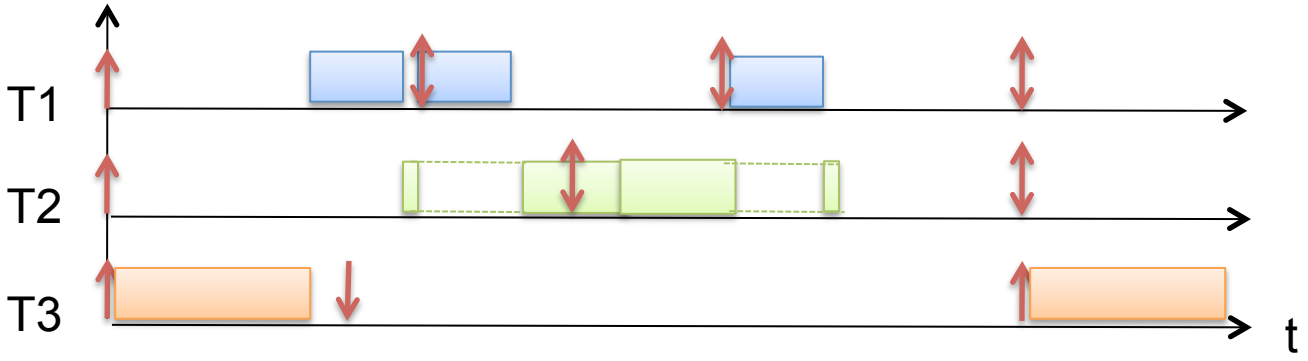
Il faut insérer des temps d'attente dans la séquence

- Comportement souhaité



Scrutation (5)

Ordonnancement préemptif



Séquence :

Le code de la tâche T2 doit être découpé en 4 morceaux

```
Code_T2(){
    instruction 1;

    instruction n;
}
```

Séquence à exécuter

```
Code séquence {
  T3();T1();T2a();T1();
T2b();T2c();T1();T2d();
}
```

Scrutation (6)

- Autre méthode
  - Exécution d'une fonction d'ordonnancement avec une période = PGCD des périodes des tâches
  - Chaque tâche est appelée par la fonction d'ordonnancement 1 fois toutes les `n` exécutions de la fonction d'ordonnancement avec `n=Période_tâche/PGCD`
- Outils synchrones utilisent ce type de méthode

Scrutation: conclusion

- Avantages

  Nécessite peu de mémoire, code simple et déterministe, temps de réaction facile à déterminer

- Inconvénients
  - Très peu flexible
  - Limité aux ordonnancements non préemptifs
  - Pas adapté au déclenchement par évènement (event trigged)

Plutôt utilisé dans les systèmes simples, lorsque les contraintes de coût sont fortes, ou lorsqu'il faut pouvoir certifier le code de l'application (fiabilité)

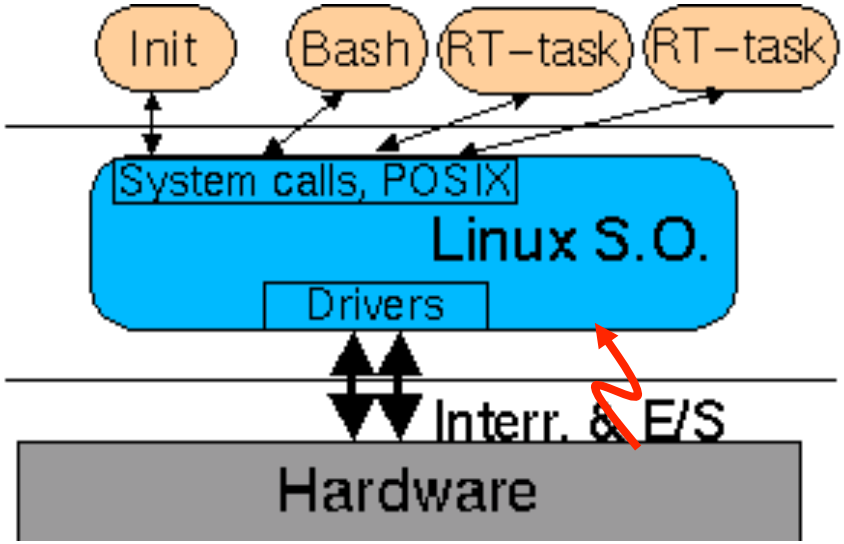
Implantation avec RTOS ← Objet de ce cours

- Avantages
  - Différents modèles de tâche:

    Tâche cyclique, tâche en attente d'1 déclencheur, tâche traitant une séquence d'événements déclencheurs, tâche traitant un choix d'événements déclencheurs (automate)
  - Différents modes d'interaction entre tâches

    ASYN/ASYN, ASYN/SYN, SYN/SYN, SYN/ASYN
  - Flexible
- Inconvénients
  - Indéterminisme, surcoût d'exécution et de mémoire

La simplicité apparente de conception et la flexibilité rendent cette approche très populaire



II. QUEL SYSTEME D'EXPLOITATION UTILISER ?

Cahier des charges d'un RTOS

- Minimum pour gérer plusieurs tâches temps réel
  - support des IT
  - support d'une base de temps
  - déclaration de tâches : 3 états principaux (*prêt, courant, en attente*)+création
  - ordonnanceur
  - éléments de synchronisation (ATTENDRE SIGNALER)
  - code chargé avec l'application (ROM/RAM)
- Surcharge minimale par rapport à un processeur nu

Services supplémentaires

- Orienté gestion tâches
  - Gestion de la mémoire
  - Structuration des échanges avec le monde extérieur (E/S, IT) : pilote de périphérique (driver)
  - outils de synchronisation et communication supplémentaires
- Orienté IHM
  - rajouter des services
  - intégrer des services temps réel dans un OS « standard »

Architecture de développement I

- Développement croisé

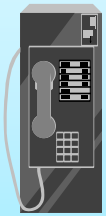
Machine de développement

•écriture des sources
•compilation croisée
(cross compilation)
•éditions des liens :
-code
-bibliothèque T.R et noyau
-chargeur

•mise au point
par simulation

•téléchargement
•console de déboguage

Machine cible



•kit de développement
•système embarqué
•E/S
•Procédé
•Clavier/Ecran parfois

Architecture de développement II

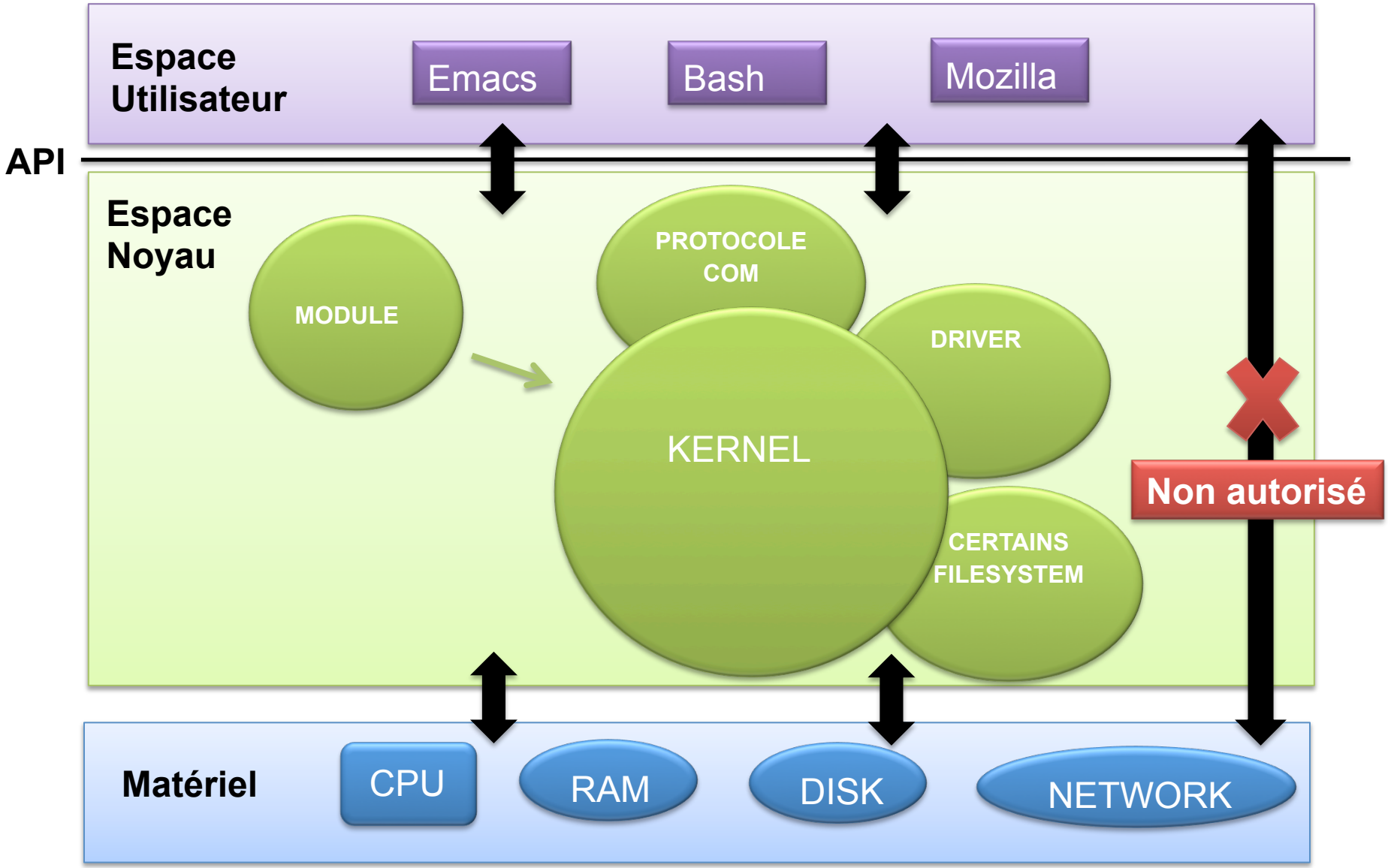
- IDEM développement croisé, mais la machine de développement est aussi la cible

III. UN OS MULTITÂCHES : LINUX

LINUX quelques propriétés

- Linux est un système d'exploitation libre
- Gère les E/S standards
  - Disque
  - IHM (clavier, souris, écran)
  - Réseaux
- Multitâches
  - Equilibre
  - Utilisation des ressources
  - Temps de réponse moyen

Linux, son noyau, ses modules



Le noyau linux (KERNEL)

- fonctions
  - ordonnancement, ses fonctions et son système de priorité
  - gestion des processus
  - gestion du temps système
  - accès aux ressources matérielles
  - gestion des interruptions
  - timers de signaux ALARME
  - files de tâches
  - gestions des identificateurs des utilisateurs et des groupes
  - gestion des modules
  - fonctions de trace, de debug (printk)
  - cas d 'instabilité système
  - accès aux informations système

Le noyau n'est pas multitâche et il est non préemptible

Les modules

- Pourquoi des modules ? **Modularité !!!**

  ⇨permet d'éviter de recompiler le noyau lors de l'ajout de drivers ou de consommer de la mémoire sur des modules non utilisés et intégrés au noyau

- un module peut utiliser toutes les fonctions, accéder à des variables et structures du noyau
- code exécuté avec niveau maximal de privilèges
- Un module peut accéder aux E/S et exécuter des instructions privilégiées

Les exceptions sont fatales au système lorsque provoquées dans un module

Modules : quelques principes

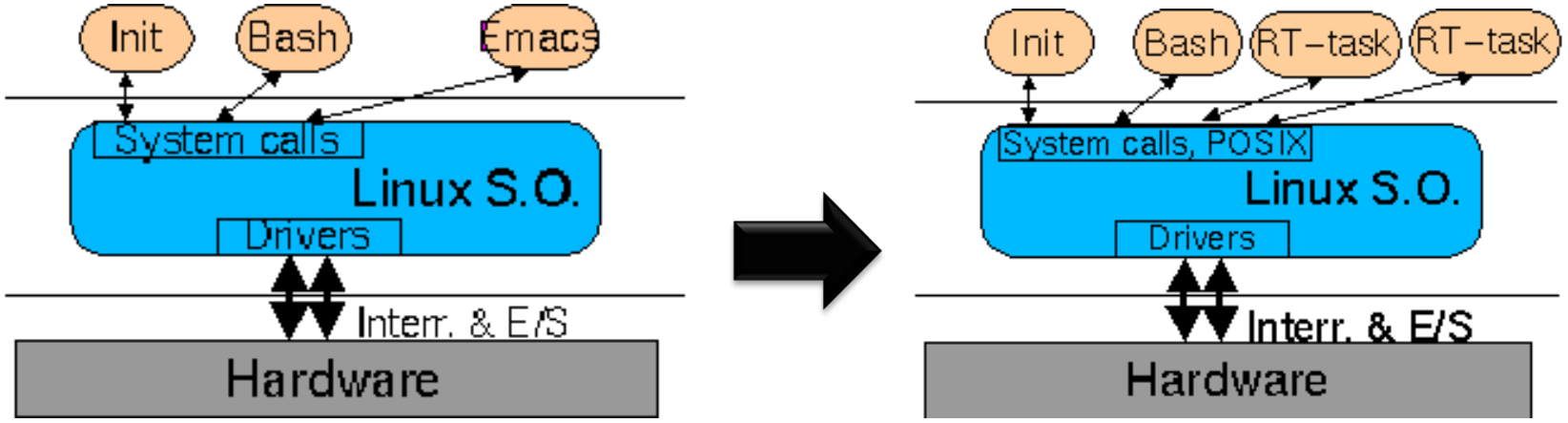
- Modules référencés dans une liste maintenue par le système (accès par `lsmod`)
- Un module ne peut appeler que les fonctions exportées par le noyau (pas n'importe quelle bibliothèque)
- Module chargé en mémoire que par superviseur (root) ou s'il en a donné le droit (`sudo`)
- Le noyau maintient un compteur de liens sur le module (pour savoir quand on peut le détruire)

Linux et le temps réel

- Processus
  - Date de réveil non garantie
  - Ordonnanceur basé sur le principe du tourniquet : répartir le temps d'exécution de manière équitable
  - noyau (appels systèmes) non préempté
- Mémoire
  - Swap sur la Ram
  - Swap sur les process
- Interruptions
  - dévalidées régulièrement par le noyau

IV. LINUX TEMPS RÉEL

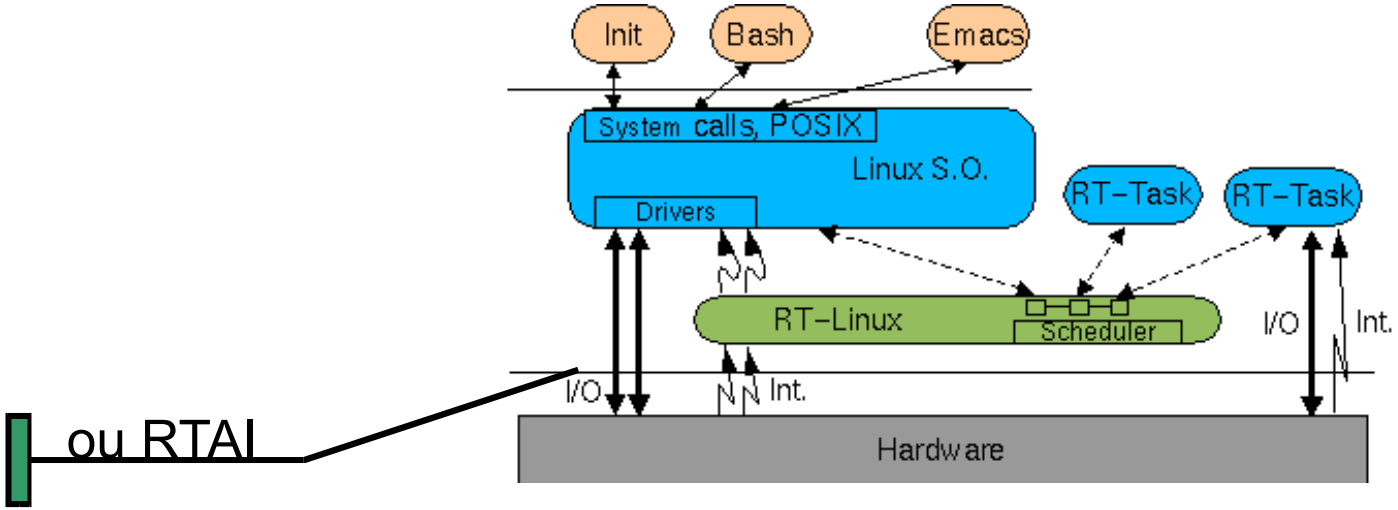
Implémentation KURT



Linux classique **KURT**

- rajouter une API « temps réel » dans le noyau
- Rendre le noyau préemptif
- Dévalider les IT le moins longtemps possible
- Temps réponse ≈ qq 100µs

Implémentation RT-Linux, RTAI



- Ajout d'un micro-noyau
- Linux exécuté en tâche de fond au même titre qu'une RT-Task
- Cohabitation Linux / exécutif temps réel
- Distribution fichiers : *https://www.rtai.org/*
- Temps réponse < 20µs

Adapté aux applications à contraintes TR strictes

Autres solutions autour de LINUX

- RTLinux
- RTAI (variante de RTLinux)
- eCos
- . . .
- et leurs distributions commerciales
  - FSMLAbs (RTLinux)
  - MontaVista HardHat
  - Lineo Embedix Realtime
  - TimeSys Linux/RT
  - LynuxWorks BLueCat Linux (Lynx Real Time Systems)

A LIRE

- Articles de Ismaël Ripoll
  - http://www.linuxfocus.org/Francais/May1998/article44.html
  - http://www.linuxfocus.org/English/July1998/article56.html
- http://kadionik.developpez.com/cours/systeme/linux-realtime/
- « Linux embarqué », Pierre Ficheux, edition Eyrolles

RTAI en quelques mots

- Exécutif léger temps réel
- Intègre au fur et à mesure de nouvelles fonctionnalités
  - multi ordonnanceurs
  - communications
  - sources disponibles
- Découpé en différents modules fonctionnels
  - on ne charge que ce dont on a besoin
  - la philosophie des créateurs de RTLinux est de garder un noyau minimal, pas chez RTAI
- support de l'unité arithmétique : FP

RTAI vit avec LINUX

- Linux est fonctionnellement inchangé
  - offre tous les avantages et la puissance des environnements modernes
- Séparation du temps réel (RTAI) et du non temps réel (Linux) :
  - tâches RT : pas d'appel système pour les services
  - Communication spéciale avec tâches non temps réel
    - Lecture/écriture mémoire
    - files d 'attente (fifo) temps réel, sorte de pipe avec propriétés spéciales
- Remarque : sans contrainte « temps réel », il existe des équivalents plus sûrs (voire simples) en Linux de tous les services systèmes que nous allons étudier

V. DÉVELOPPER AVEC RTAI

Comment intégrer une application RT ?

- Modifier le noyau
  - installation de RTAI, suivez le guide
    - patcher Linux
    - compiler un nouveau noyau
    - installer les modules
- Ecrire l'application sous forme de modules
  - connecté directement au noyau (espace kernel)
  - incorpore tâches RT
  - 1 ou plusieurs module
- Exécuter l'application
  - Compilation des modules applicatifs
  - Chargement des modules RTAI
  - Chargement des modules applicatifs

Développer un module (linux 2.4)

```
#define MODULE
#include<linux/init.h>
#include <linux/module.h>
#include<linux/version.h>

static int output=1;

int init_module(void) {
   printk("Output= %d\n",output);
   return 0;
}

void mon_code(void) {
  output++;
}

void cleanup_module(void){
   printk("Adiós, Bye, Ciao, Ovuar, \n");
}
```

Développer un module (linux 2.6)

```
#define MODULE
#include<linux/init.h>
#include <linux/module.h>
#include<linux/version.h>     plus nécessaire

static int output=1;

int mon_init(void) {
   printk("Output= %d\n",output);
   return 0;
}

void mon_code(void) {
  output++;
}

void mon_cleanup(void){
   printk("Adiós, Bye, Ciao, Ovuar, \n");
}
module_init(mon_init);
module_exit(mon_cleanup);
```

Comment mettre en œuvre l'application ?

- Compiler le module :
  - Linux 2.4 : `gcc -I /usr/src/linux/include/linux -O2 -Wall D__KERNEL__ -c exemple1.c`
  - Linux 2.6 : nécessite les sources du noyau + makefile
- Charger le module

```
> insmod exemple1.ko (exemple1.o sous noyau 2.4)
```

- Printk écrit sur la console

```
> dmesg | tail -1
Output= 1
```

- Liste des modules actifs

```
# lsmod
Module     Pages      Used by:
exemple1       1           0
sb             6           1
```

Etapes du chargement (résumé)

1) Vérification des symboles externes référencés
2) Vérification de version de noyau
3) Allocation mémoire noyau pour contenir le code et les données du module (mémoire physique)
4) Copie du code du module dans l'espace alloué
5) Maintenance de la table des symboles du module (`output` et `mon_code`)
6) Exécution de la routine d 'initialisation du module `init_module` (point d'entrée)

Comment nettoyer le système ?

- Enlever le module
  - exécutera la fonction cleanup_module()

```
# rmmod exemple1
# dmesg | tail -2
Output= 1
Adiós, Bye, Ciao, Orvua,
```

Passer des données au module ?

- En affectant des valeurs aux variables globales

```
# insmod exemple1.ko output=4
# dmesg | tail -3
Output= 1
Adiós, Bye, Ciao, Orvua,
Output= 4
```

- Toutes ces commandes peuvent être regroupées dans un makefile et/ou dans un fichier de commande

Makefile

- Interprété par la commande `make`
- Ensemble de **règles**
- Règle : dit comment exécuter une série de commande pour construire une **cible** à partir de fichiers sources. Définit aussi les **dépendances** (fichiers d'entrées) nécessaires aux commandes

```
règle : dépendances
        commande
```

- www.gnu.org/software/make/manual

Modules de RTAI

- `rtai_hal` : fonctions de base
- `rtai_sched` : ordonnanceur mono processeur
- `rtai_sem` : gestion des sémaphores
- `rtai_fifos` : interface « fichier » entre modules RT et processus standards

+ vos modules applicatifs

- liaison matériel
- définition de tâches

- A charger au démarrage de chaque lancement d ’une application RT

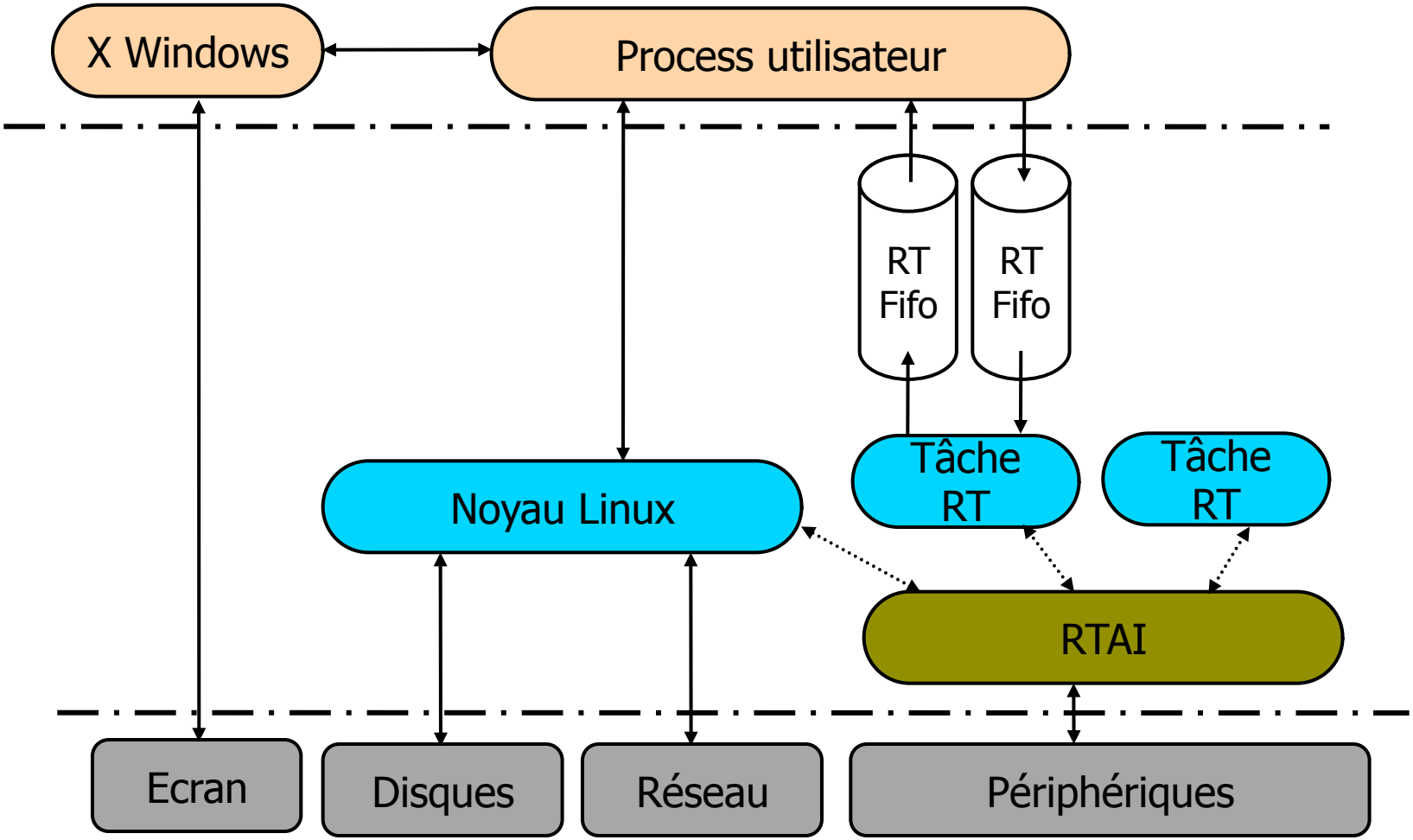
⇨ script de commandes

Exemple script lancement

- Utilisation des commandes linux
- rt_modprobe

Structure d'une application

- Découpler partie temps réel / fonctions non temps réel



VI. TÂCHES RT ET ORDONNANCEMENT AVEC RTAI

Tâches et organisation sur le processeur

- Définition
  - Une tâche est une unité de travail séquentielle qui peut être exécutée sur un processeur, composée de :
    - 1 programme
    - 1 pile d ’exécution
    - 1 zone de données
    - registres particuliers processeur (compteur ordinal, pointeur de pile) : sauvegardés lors du changement de tâche
  - En général et en temps réel une tâche est exécutée
    - périodiquement (sur initiative de l’horloge)
    - sur arrivée d ’un événement (boucle et attente d ’événement)

Tâches sous Linux/RTAI

- Sous Linux, 2 niveaux :
  - processus

    Intègre des mécanismes de protection

  - thread (processus léger)
    - à l ’intérieur d’1 processus peuvent coexister plusieurs threads
    - partage de la même zone mémoire commune
    - changement de contexte (de tâche) plus rapide
- Tâche RTAI = thread

RTAI : Gestion de tâches

- RTAI
  - chaque tâche RT est un thread `RT_TASK`
  - Noyau Linux(noyau) = thread de priorité *idle*

- Ordonnancement des tâches
  - par défaut
    - préemptif
    - priorité fixe ou EDF
    - risque de surcharge processeur
    - Linux : tâche de fond

Créer une tâche

```
#include <rtai_sched.h>
MODULE_LICENSE("GPL");
static RT_TASK tache;
#define STACK_SIZE 2000

iret = rt_task_init(&tache,  // pointe sur la tache
                    routine, // nom de la fonction à exécuter
                    1,       // entier passé à la routine
              STACK_SIZE,    // taille de la pile d'exécution
                    1,       // int priorité, 0=plus haute priorité
                    0,       // int 0 => n'utilise pas le fpu
                    0       // signal (voir rt_task_signal_handler)
                      );
```

- iret = 0 si ok, <0 sinon
- la création d'une tâche s'effectue dans le init_module

rt_task_init ne déclenche pas la tâche

Rendre une tâche périodique

- Dans init_module

```
iret = rt_task_make_periodic(&tache,
           date_depart, nano2count(PERIODE));
```

⇨ rend périodique, unités en comptes internes

- Dans la tâche

boucle infinie contenant le corps de la tâche

+ fonction `rt_task_wait_period ();`

⇨ suspend l'exécution jusqu'au début de la prochaine période

Suspendre et détruire une tâche

- Suspension

```
rt_task_suspend( rt_whoami() );
```

- Reprise

```
rt_task_resume( &tache);
```

⇨ Permet aussi de déclencher la tâche en mode non-périodique

- Terminaison

```
rt_task_delete( &tache);
```

Gérer le temps

- Timer : 2 modes de fonctionnement
  - `rt_set_oneshot_mode()`
    - Pentium récents utilise le CPU TSC (Time Stamp Clock) => base de temps variable
    - Sur 386,486 et premiers pentium pas de CPU TSC => lecture du timer 8254 => couteux
  - `rt_set_periodic_mode()`
    - Mode par défaut
    - Période fixe
    - Les tâches dont la période n'est pas multiple de la période du timer sont servies au mieux

Gérer le temps (2)

- Avant tout il faut démarrer le timer

```
tick_per = start_rt_timer(per);
```

  - En mode one shot : `per` ignoré
  - En mode periodic : `per` = période du timer en count,

    `tick_per` = période attribuée (count)

- Prise d'infos sur le temps

```
rt_get_time();   en unités internes
rt_get_time_ns();   en nanosecondes
```

- Conversions

```
RTIME tn, tc;
tc = nano2count(tn);
tn = count2nano (tc);
```

- Arrêter le timer

```
stop_rt_timer();
```

RTAI : Ordonnancement

- Politiques d ’ordonnancement combinables
  - EDF > priorités fixes
  - Préemptif priorité fixe : tâche de plus haute priorité depuis le plus longtemps en attente
  - Linux est vu comme une tâche de fond (thread de priorité idle)
- Priorité
  - Fixe : de 0 à `RT_LOWEST_PRIORITY`

```
ma_prio = rt_get_prio( rt_whoami());
ancienne = rt_change_prio(rt_whoami(),
                          ma_prio+1);
```

  - Dynamique

```
rt_task_set_resume_end_times(date_reveil, date_echeance);
rt_task_set_resume_end_times(-periode,-echeance);
```

RTAI : Etat des tâches

- 3 catégories
  - -1 unrunnable
  - 0 runnable : liste des tâches prêtes
  - >0 stopped
    - 1 : endormie : peut être réveillée par un signal ou l ’expiration d ’un timer
    - 2 : endormie mais non susceptible d ’être réveillée par événement
    - 4 : terminée (zombie)
    - 8 : stoppée par signal de contrôle ou par instruction de debug ptrace
    - 16 : en train d ’être paginée
    - 32 exclusive (seule à être réveillée lors d ’une attente)

Gestion SMP

- SMP : Symetric Multi Processor
- 2 solutions
  - 1 ordonnanceur gère les n processeurs
    - attribue les n threads les plus prioritaires parmi les t prêts
  - 1 ordonnanceur par processeur
    - chaque thread est attribué à un couple {processeur, ordonnanceur}
- les données/ synchros restent communes aux différents tâches
- RTAI : solutions intechangeables (modules spécifiques)

VII. SYNCHRONISER, COMMUNIQUER

Synchroniser / Communiquer: pourquoi ?

- Transmission d ’événement
  - fugitif, mémorisé, incrémenté
  - avec ou sans valeur associée
- Assurer un ordre
  - avant / après
  - nombre d ’exécution
- échanger des données
  - avec synchronisation
  - sans synchronisation
    - assurer l ’intégrité des données (de la mémoire)

RTAI : sémaphores binaires - mutex

- Rendez-vous, exclusion mutuelle
- Création

```
SEM mon_sem;
rt_typed_sem_init(&mon_sem,  // pointeur sémaphore
                          1, // valeur initiale
                  BIN_SEM    // type de sémaphore
                          );
```

- Prendre le sémaphore

  Si le sémaphore est présent : on prend le sémaphore et on continu

  Si le sémaphore est absent : on est bloqué jusqu'à ce que le sémaphore soit restitué

```
ierr = rt_sem_wait(&mon_sem);
```

- Restituer le sémaphore

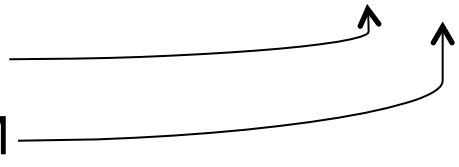
```
ierr = rt_sem_signal(&mon_sem);
```

Sémaphores (2)

- Problèmes : inversion de priorité, interblocage
- Autres types

```
rt_typed_sem_init(&mon_sem,1,BIN_SEM);
```

  - A compte : CNT_SEM
  - Ressource : RES_SEM
    - met en place le protocole de priorité plafond



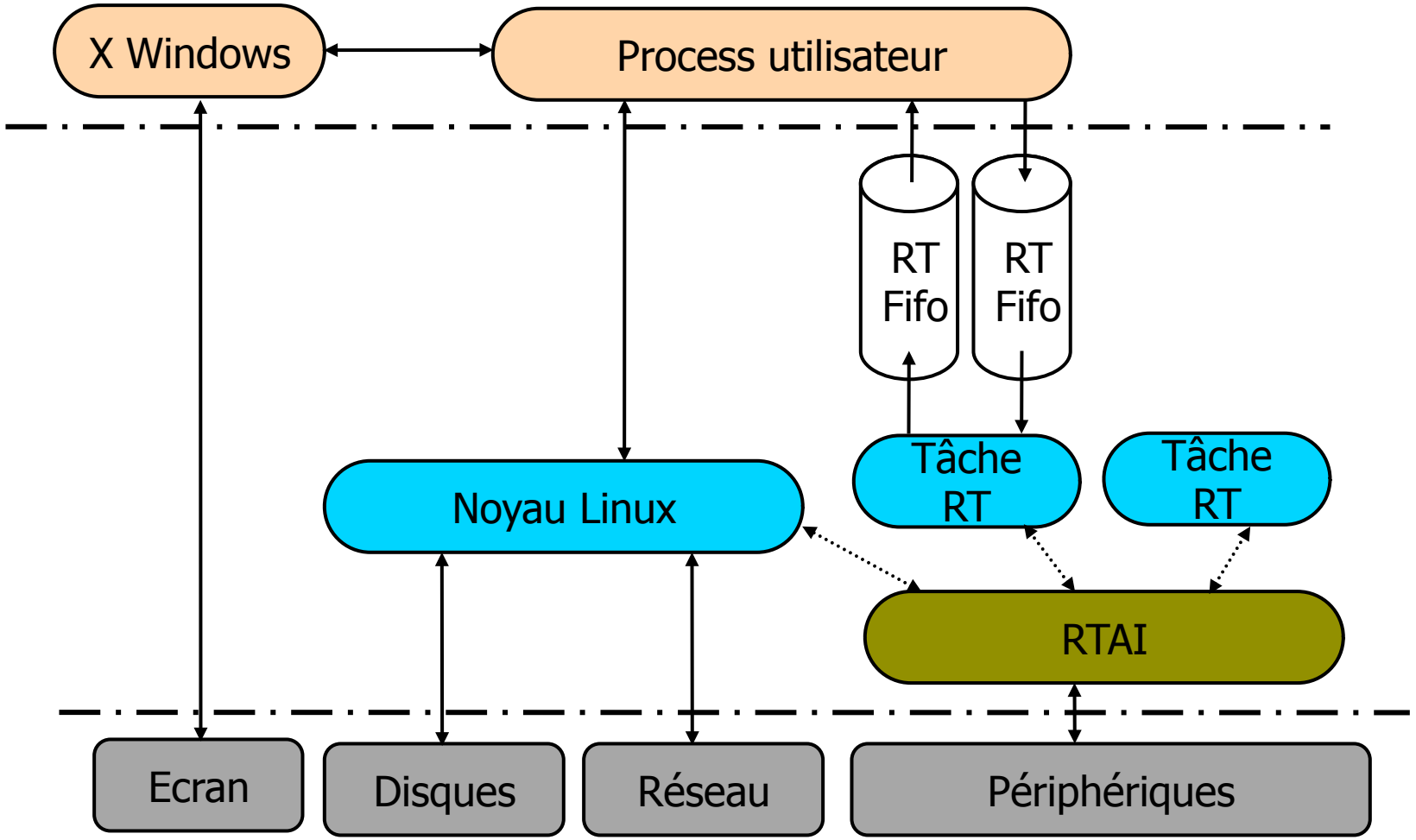
- Destruction

```
- ierr = rt_sem_delete(&mon_sem);
```

- Variantes asynchrones

- `rt_sem_wait_if` : pas de blocage si sémaphore absent
- `rt_sem_wait_until(&sem,date)` : blocage au maximum jusqu'à `date`
- `rt_sem_wait_timed(&sem,durée)` : blocage au maximum pendant `durée`

RTAI : Fifos


X Windows
Process utilisateur
RT Fifo
RT Fifo
Tâche RT
Tâche RT
Noyau Linux
RTAI
Ecran
Disques
Réseau
Périphériques

RTAI : RTFifos coté process utilisateur

- côté process utilisateur
  - périphérique spéciaux de type caractère :
    - /dev/rtf0, /dev/rtf1 v…
    - accessibles à travers les appels : open, ioctl, read, write
  - communication sans structure
    - file FIFO tampon d ’un nombre d ’octets fixé
  - n ’existe (ne peut être ouvert) que si déjà créé par tâche RT ou par appel spécifique

RTAI : RTFifos coté RT

- 2 modes d ’utilisation
  - comme périphérique
  - API spécifique RT

`rtf_create(unsigned int fifo, int taille);`

création,

périphérique `/dev/rtf`***`fifo`*** existe,

`rtf_destroy(unsigned int fifo);`

destruction,

mémoire réallouée

`rtf_put(fifo, char *tampon, int compte);`

`rtf_get(fifo, char *tampon, int compte);`

retour -1, si insuffisamment de données ou

de place

non bloquant

RTAI : RTFifos

- côté tâche RT (2)
  - API spécifique RT

```
int my_handler(unsigned int fifo);
```

```
rtf_create_handler(3,&my_handler);
```

routine `my_handler` exécutée en mode noyau quand des données sont lues ou écrites

  - permet de réveiller une tâche RT, via p.ex. un task_resume
  - permet d ’éviter la scrutation régulière de la FIFO
  - permet de mettre en place un schéma ASYN/SYN quelconque avec sémaphores

`rtf_resize =>` redimensionner la fifo

- plus d ’autres fonctions

Messages

- Caractéristiques
  - Communication point à point (tâche à tâche)
  - message : valeure entière
  - SYN/SYN
  - `ierr = rt_send(&tache_but,valeur); //SYN`
  - ```
    ierr = rt_receive
           (&tache_orig,&entier); //SYN
    ```
- Variantes
  - `If :` send ASYN, pas d'envoi si recepteur non prêt, pas de blocage de l'emetteur
  - `timed et until :` blocage limité dans le temps
  - `rt_receive(0 … :` n 'importe quel émetteur
- Pas d 'initialisation spécifique
    ```

Boîtes aux lettres (Bal)

- Caractéristiques
  - Communication multi/multi point
  - message : taille variable
  - ASYN/SYN

  - `ierr = rt_mbx_send(&Bal,&msg,taille_msg);` ASYN, mais blocage tant que le message entier n'a pas été copié dans la Bal
  - `ierr = rt_mbx_receive(&Bal,&msg,taille_msg);` Blocage tant que le message n'a pas été reçu

- Variantes send
  - `if` : envoi ssi le message entier peut être copié dans la bal
  - `Wp` : envoi autant d'octets que possible sans blocage
  - `timed et until : blocage limité dans le temps`
- initialisation spécifique
  - `ierr = rt_mbx_init(&Bal,taille)`
  - `ierr = rt_mbx_delete(&Bal);`

Le problème de l ’allocation mémoire

- Segmentation,
- Mémoire paginée
- pas new/delete ou malloc/free
  - kmalloc / kfree

- **non temps réel**

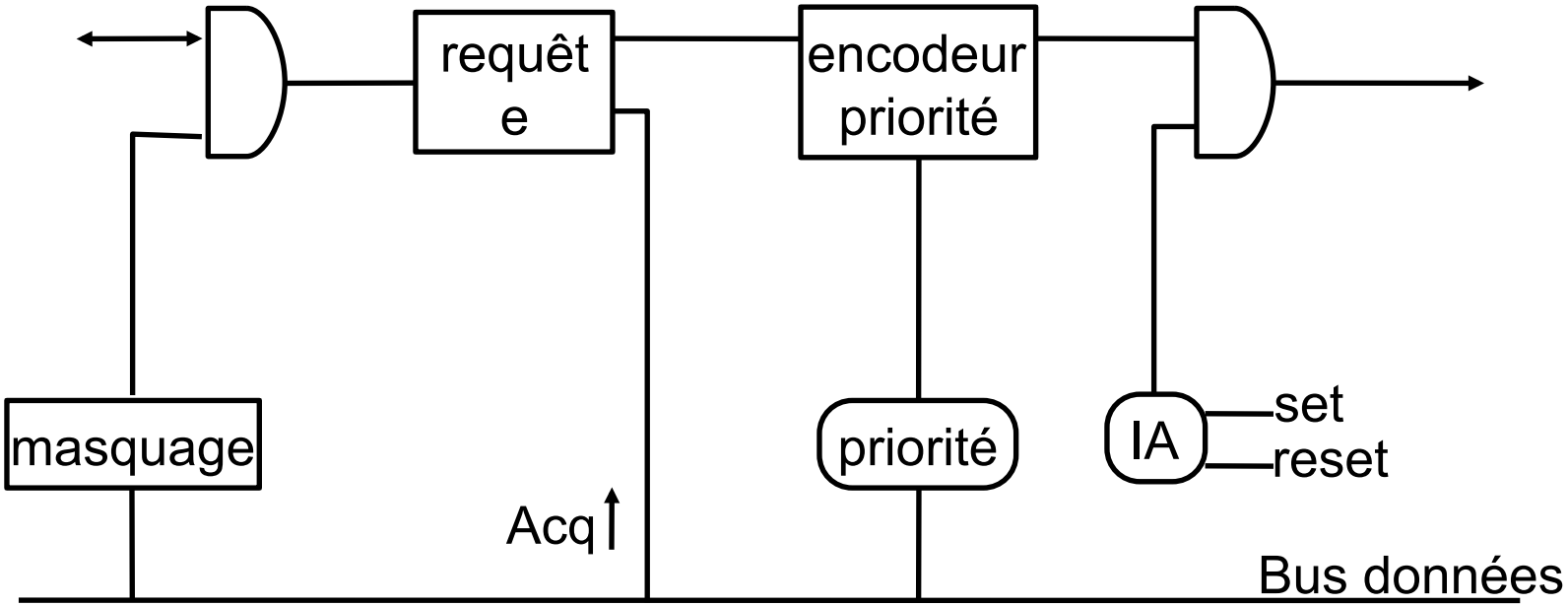
- **solutions RT :**
  - s ’allouer la mémoire de manière statique
  - allocation dans une table de blocs limitée

- utilisation de kmalloc au niveau noyau

RTAI : Interruptions

- permet réactions rapides
- inhibe le noyau temps réel
- niveau matériel et niveau logiciel
- gestionnaire d ’IT simples avec temps de réponse fixe (ou borné)
  - prise en compte d ’un événement
  - phase urgente, laisser une tâche dans l ’espace utilisateur faire le travail de fond
- ITs/événements peuvent être émis par :
  - équipement informatique (clavier, écran, disque …)
  - équipement industriel (capteur, actionneur, alarme …)
  - horloge : donne le rythme (tick = période d échantillonnage du système)

Interaction par Interruption



- rt_global_cli() rt_global_sti()
- rt_disable_irq(), rt_enable_irq()

RTAI : Interruptions (2)

- Installation

```
#define IT_CLAVIER 1
 rt_request_global_irq(IT_CLAVIER,
 mon_gestionnaire);

  rt_free_global_irq(IT_CLAVIER);

void mon_gestionnaire(void) {
 rt_pend_linux_irq(IT_CLAVIER);
```

- Utilisation
  - traitement urgent
  - réveiller tâche endormie

RTAI : communication avec le matériel

- par pilote de périphérique (`posixio`)
  - « fichiers » `/dev/nom`
    - open/ close/ read/ write
- accès direct aux ports E/S

```
#include <asm/io.h>

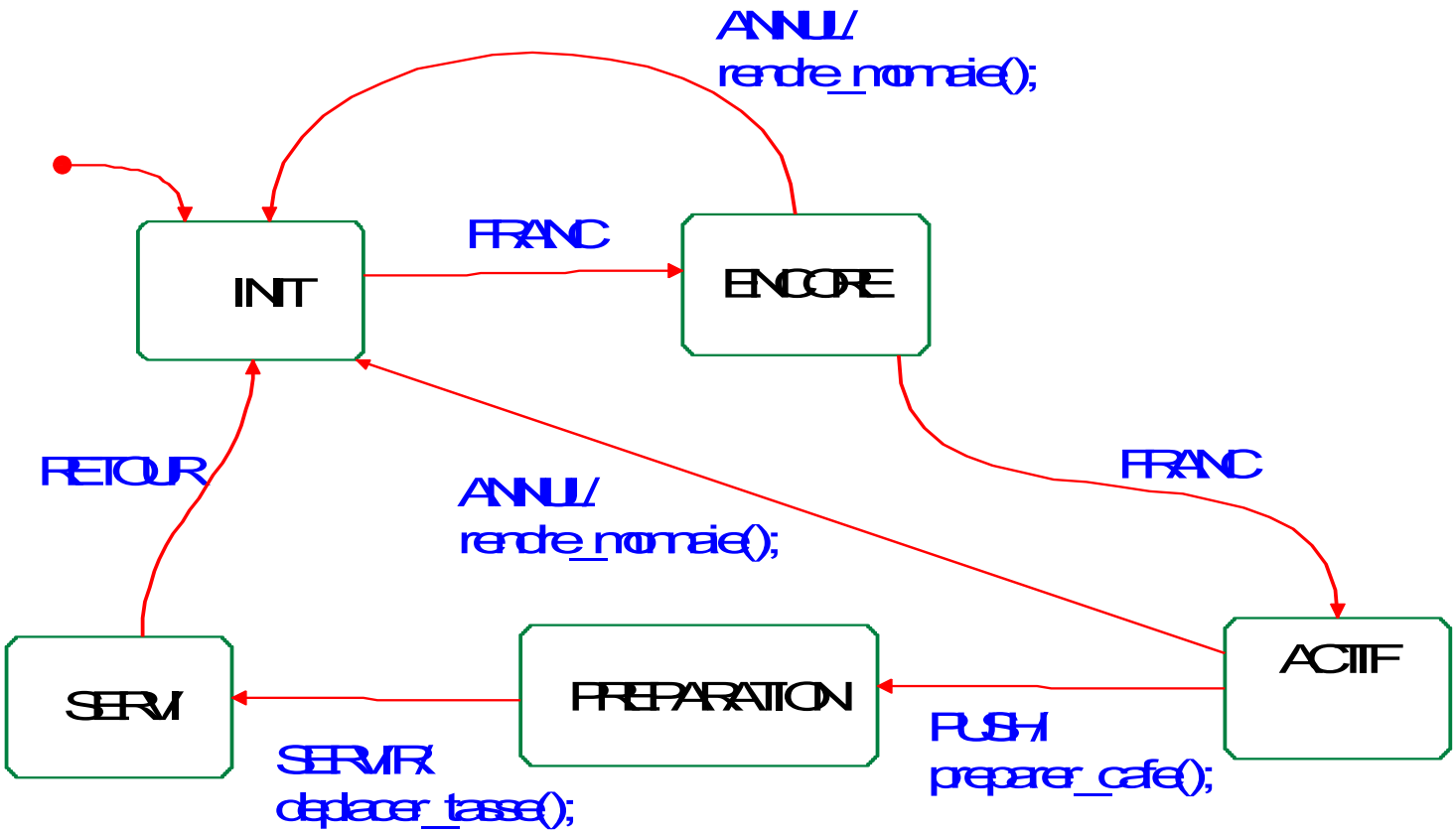

outb(valeur, adresse)
valeur = inb(adresse)
```

    - taille du mot : b, w, l
    - pause : _p : outl_p

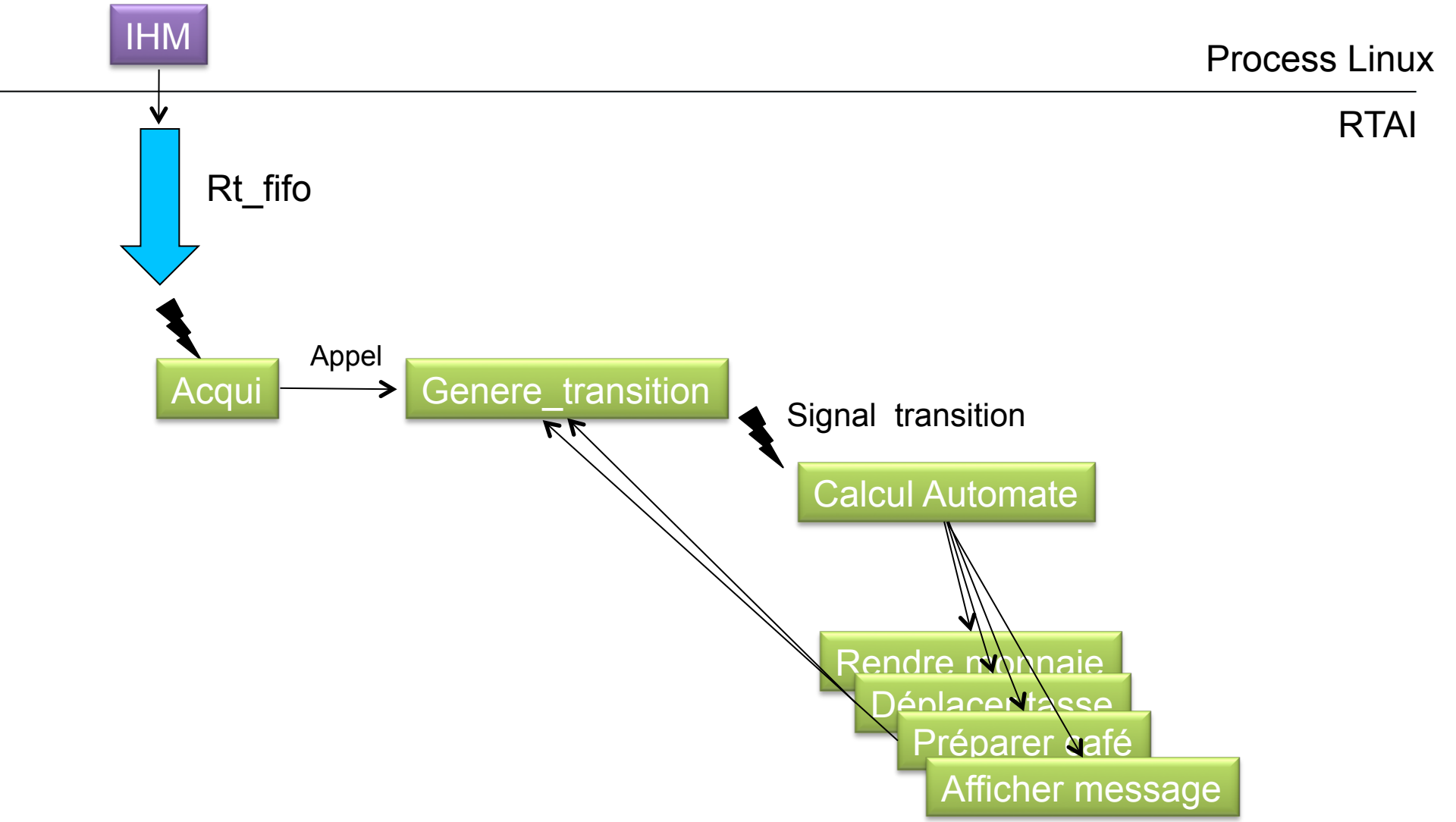
Schémas classiques

- Chien de garde (Watchdog)
- Comment gérer un automate d ’état (machine à café, menu téléphone …)
- BD : lecteur rédacteur
- Client Serveur
- tableau noir
- …

Exemple : machine à café



Machine à café : spcéfication fonctionelle



CONCLUSION

RTAI : quelques remarques

- évolution permanente
- possibilités en phase de mise au point non détaillées ici
  - outils de trace : LTT
  - exécution en mode user : LXRT
- Xenomaï
  - POSIX, VxWorks,VRTX,pSOS+,µITRON,RTAI

RTAI : quelques remarques

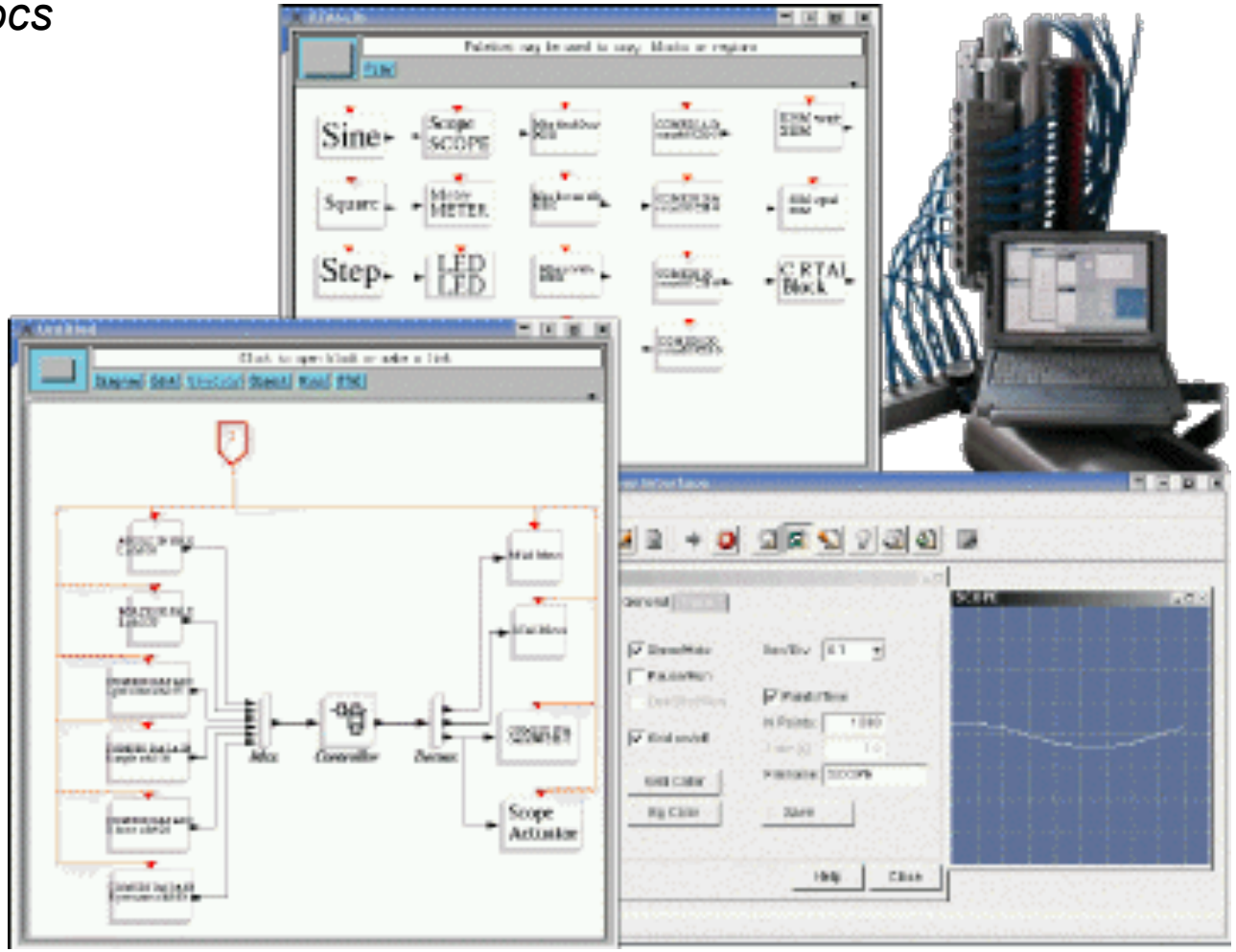
- évolution permanente
- possibilités en phase de mise au point non détaillées ici
  - debug
  - outils de trace : LTT
  - exécution en mode user : LXRT
- Xenomai

RTAI-LAB

Spécification schémas-blocs
SCILAB/SCICOS
MATLAB/SIMULINK

↓ Génération code + Librairie RTAI-LAB

RTAI
Exécution



http://www.rtai.org
http://www.scicos.org/
http://www.mathworks.com/products/simulink/

ET LES AUTRES RTOS …

Quelques OS … (source http://www.dspconsulting.com/rtos.html

RTOS	URL	Processeurs	Licence
Tornado, VxWorks	http://www.wrs.com	x86, 68k, PPC, CPU 32, i960, SPARC, SPARCLite, SH, ColdFire, R3000, R4000, C16X, ARM, MIPS	$16,500 par poste + cible
Hard Hat Linux	http://www.mvista.com	x86, PPC, MIPS and ARM	Non
VRTX	http://www.mentor.com	x86, 68k, PPC, i960, ARM	$2,000 par poste + cible
OS-9 V3.0	http://www.microware.com	x86, 68k, PPC, SH3, StrongARM	? + cible
QNX	http://www.qnx.com	AMD ÉlanSC300/ 310/ 400/ 410, AM386 DE/SE Intel386 EX, Intel486, ULP Intel486, Pentium, Pentium Pro, and NatSemi NS486SXF.	Prix calculé sur les modules utilisés sur la cible et par unité vendues
pSOS	http://www.isi.com	86, 68k, PPC, 683xx, CPU32(+), MIPS R3000/4000/5000, Coldfire 510x/520x, i960, ARM 7 (TDMI), SH1/2/3, M32R	$17,000 par poste + cible
RTX	http://www.vci.com	x86, P5, P6	$150/RTX licence +cible
LynxOS	http://www.lynx.com	x86, 68k, PPC, microSPARC, microSPARC II, PA-RISC	US$10,000 par poste Licence: US$7000 + cible

…. + beaucoup d'autres

Choix d'un RTOS

- Performances = temps pour rendre les services à l'application, depend de:
  - la plateforme matérielle (processeur, carte, horloge,..), du compilateur et de sa conception
  - Quelles mesures ? Protocoles de mesures ? => benchmarks
  - Temps de préemption, Temps activation Task, sémaphore, allocation mémoire, services …
- Prix : licence + royalties sur chaque cible
- Accès au Code Source
- Environnement de développement
- Panel de cibles