RTAI

Premiers pas

RTAI

Real-Time Linux Introduction

- Linux is a free Unix-like operating system that runs on a variety of platforms, including PCs. Numerous Linux distributions such as Red Hat, Debian and Mandrake bundle the Linux OS with tools, productivity software, games, etc.
- The Linux scheduler, like that of other OSes such as Windows or MacOS, is designed for best average response, so it feels fast and interactive even when running many programs.
    - However, it doesn't guarantee that any particular task will always run by a given deadline. A task may be suspended for an arbitrarily long time, for example while a Linux device driver services a disk interrupt.
    - Scheduling guarantees are offered by real-time operating systems (RTOSes), such as QNX, LynxOS or VxWorks. RTOSes are typically used for control or communications applications, not general purpose computing.
- Linux has been adapted for real-time support. These adaptations are termed "Real-Time Linux" (RT Linux).
- Numerous versions of RT Linux are available, free or commercial. Two commonly available free RT Linux versions are
    - the Real-Time Application Interface (RTAI), developed by the Milan Polytechnical University and available at www.aero.polimi.it/~rtai/
    - RTL, developed by New Mexico Tech and now maintained by FSM Labs, Inc., with a free version available at www.rtlinux.org.

RTAI

“These RT Linux systems are **patches** to the basic Linux kernel source code”.

-> bonne connaissance de linux
-> bases de l’architecture du noyau
-> programmation du noyau
-> compilation des sources
-> notion de modules : chargement dynamique, développement (services drivers, etc...)

RTAI (Real Time Application Interface) est une interface temps réel développée comme un ensemble de modules du noyau Linux qui permet d'accéder à des fonctionnalités temps réel que ne permet pas un Linux standard :

- préemptions
- gestion des interruptions
- ordonnancement
- Timers précis et temps réels

RTAI

Une fois installé, RTAI est un module dormant du noyau Linux.

lors de l'initialisation, le module :

- initialise ses variables de contrôle
- fais une copie des adresses de la table d'IRQ de Linux
- initialise les fonctions spécifiques de gestion des interruptions (structure RTHAL)

RTAI a aussi la capacité d'interfacer les horloge 8284 et APIC (utilisées avec le processeurs).

a part ça, il ne fait rien.....jusqu'à ce qu'on monte le module

RTAI

Une fois activé, RTAI

- met en place les gestionnaire d'horloge matérielle

-verrouille tous les CPU

- intercepte les signaux matériels et le redirige vers ses propres gestionnaires (tables, adresses, fonctions de gestion, etc) dans une structure unique : RTHAL

- déverrouille le matériel

A ce point, Linux est toujours opérationnel, mais n'as plus le contrôle de la machine (vous en ferez l'expérience...).

Linux est en fait alors une tache administrée par RTAI, au même titre que les autres taches (celles précisément dont vous allez demander une exécution temps réel).

-> RTAI préempte le noyau.

RTAI

Pour satisfaire le fonctionnement d'application temps réel, RTAI implémente sous forme de modules additionnels :

- une gestion propre de la mémoire partagée
- des FIFO (/dev/rtfxx) et des sémaphores
- une gestion des messages (mailbow, messages, etc..)
- des ordonnanceurs temps réels
- une API POSIX
- gestion de la virgule flottante

RTAI

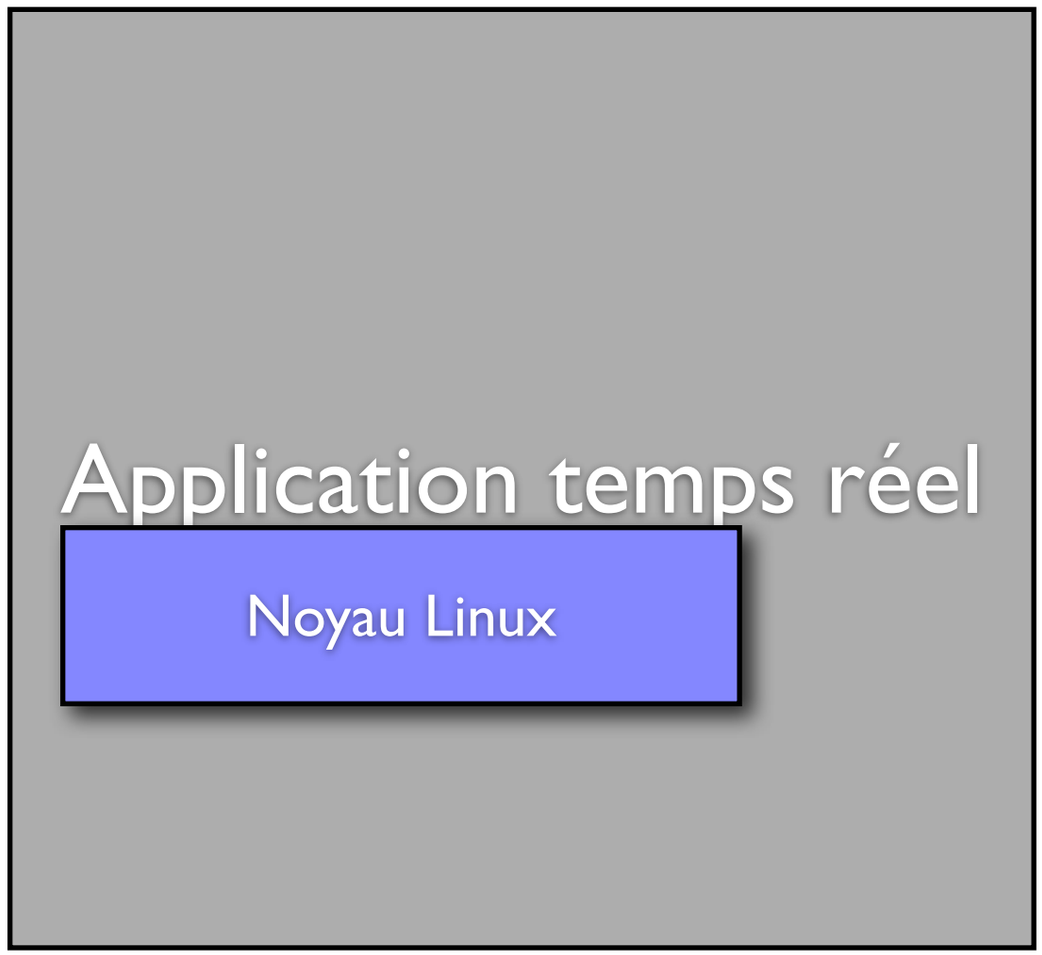
INSTALLATION

RTAI

Objectif :

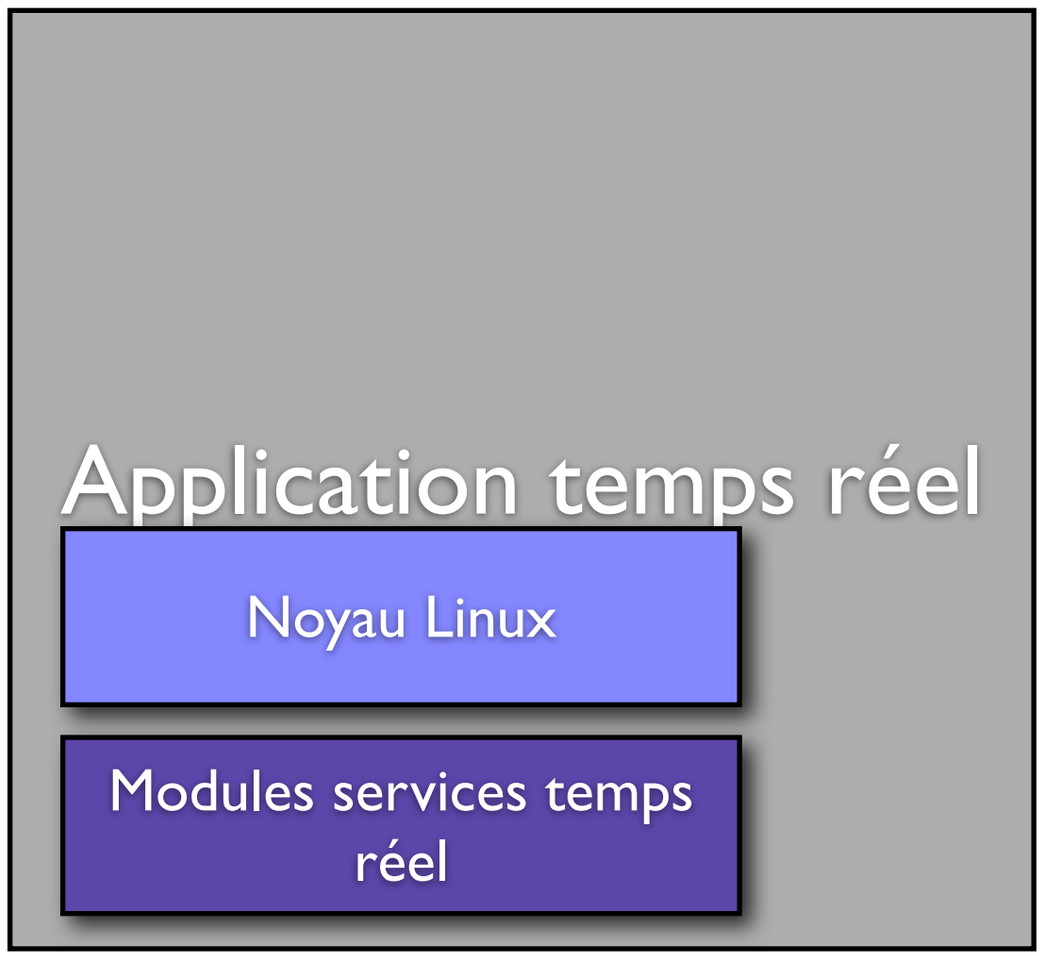
RTAI

Objectif :



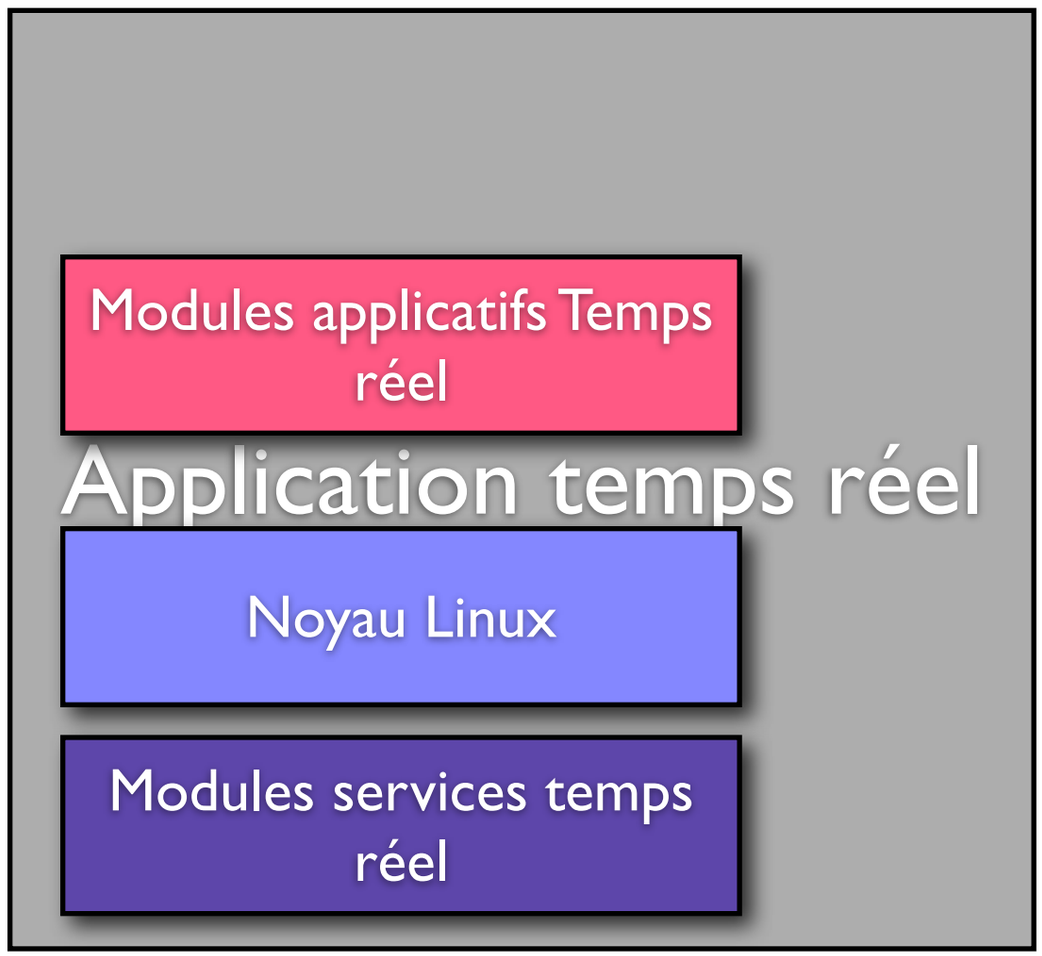
RTAI

Objectif :



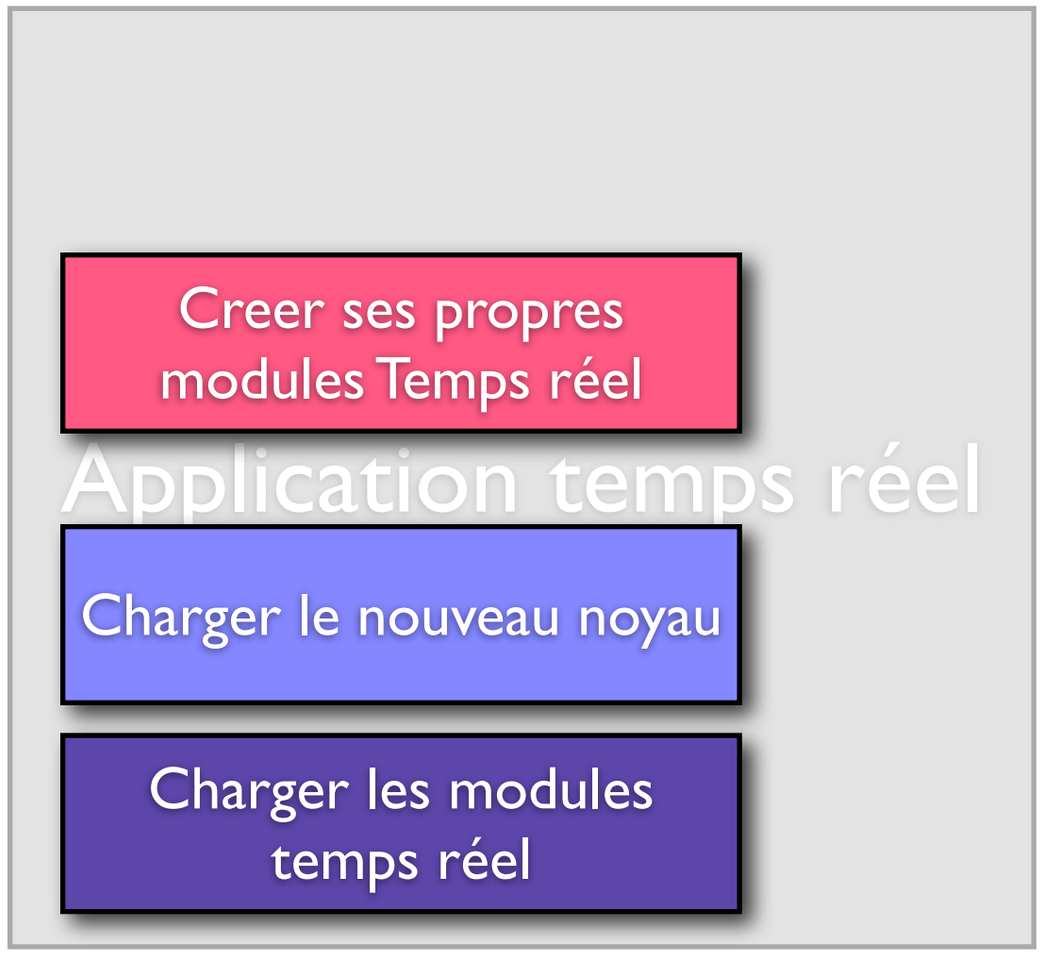
RTAI

Objectif :



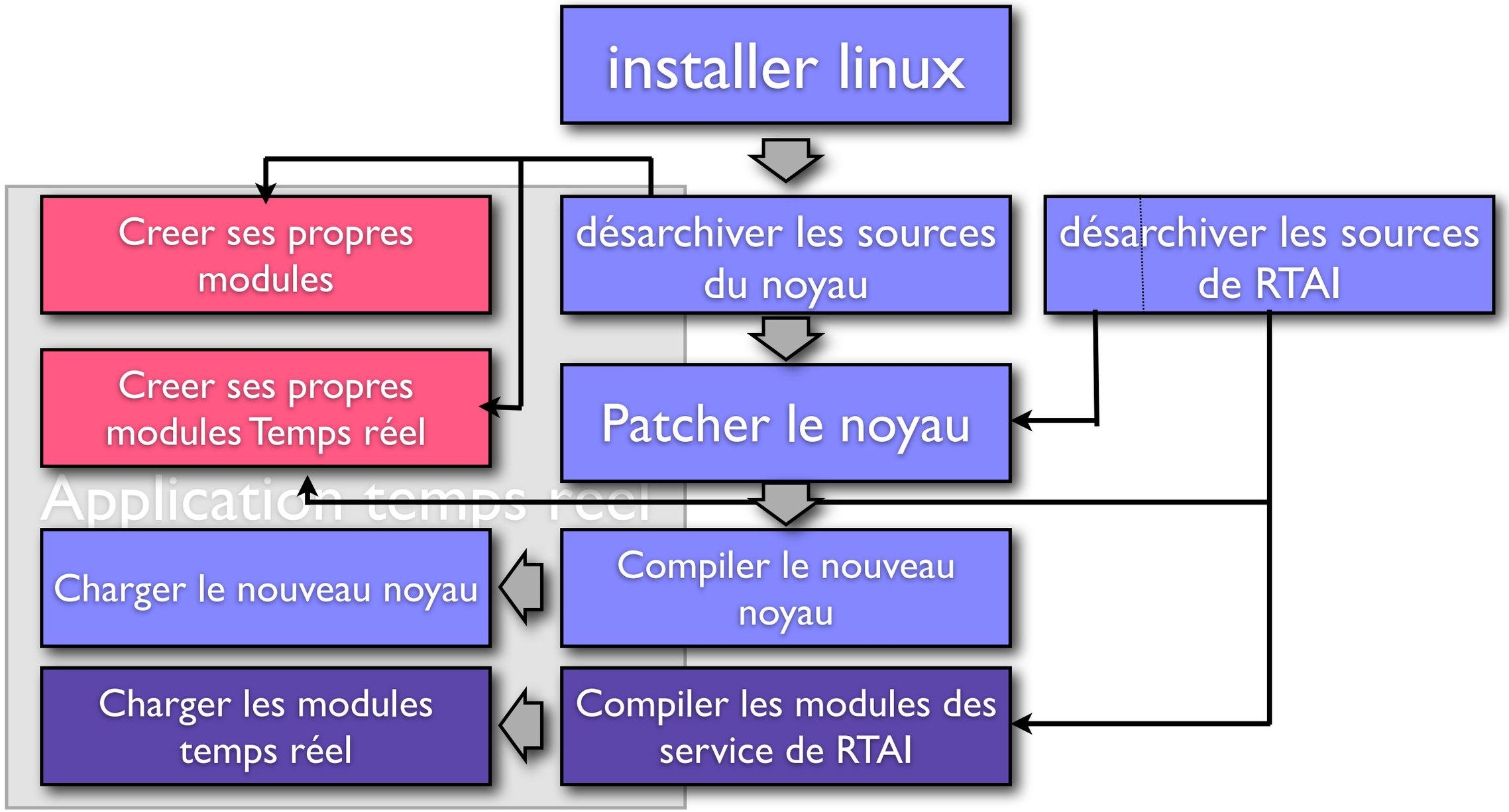
RTAI

Vue d'ensemble :



RTAI

Vue d'ensemble :



RTAI

INSTALLATION :

installer un OS Linux standard

RTAI

installer linux

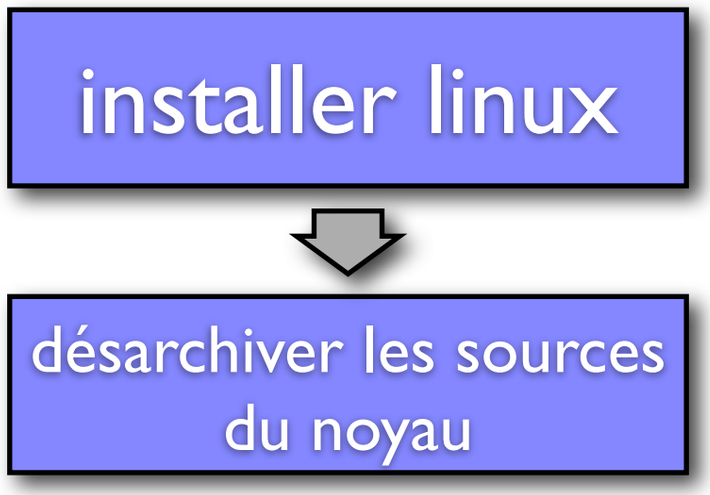
Nous partirons de Linux Ubuntu “Hardy Heron” 8.04
Distribution de 2008, **éprouvée**.
Stable : connue pour sa stabilité long terme (**LTS**) : 10 ans de support par ubuntu
Noyau “Récent” (2.6.24)

- Installation “classique”
- creer les partitions / , /boot , /usr, /home serait un plus
- Suivre les instructions du programme d’installation
- verifier que les packages de développement standards sont présents :
- editeur, includes, libc, gcc, make, etc... (c’est votre distribution de travail, a vous de la configurer)
- installer les sources du noyau
- tester un “hello world” standard en C (avec makefile rudimentaire)

RTAI

Adapter le noyau Linux pour interfacer les services de RTAI

RTAI

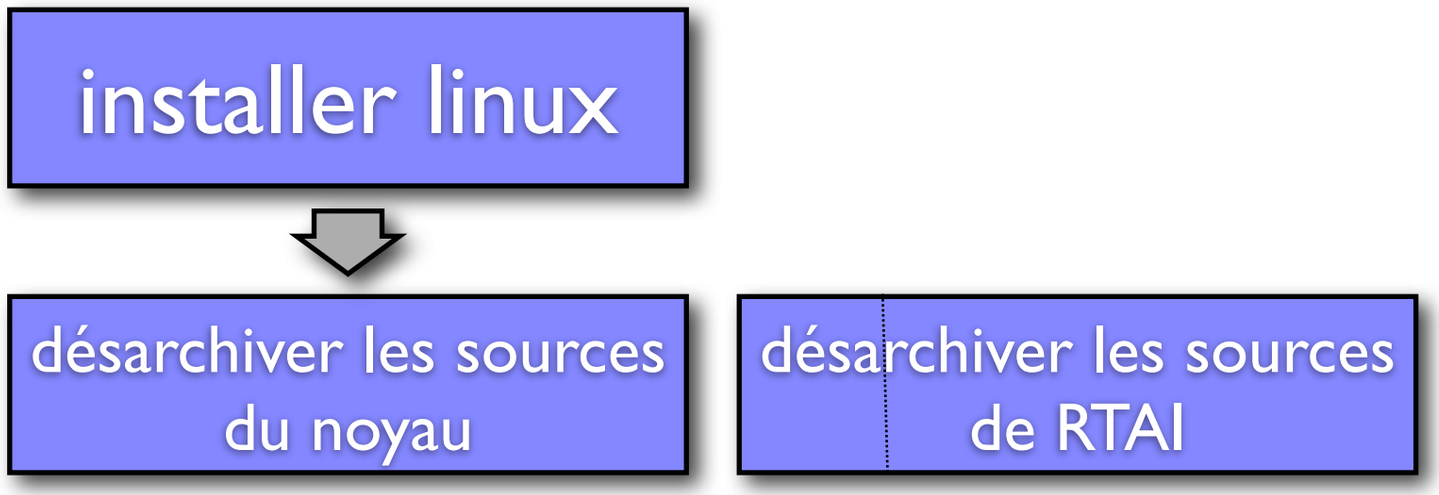


- Préparation : Installer les packages : **linux-kernel-devel fakeroot kernel-wedge kernel-package git-core**
- Copier sources du noyau 2.6.24.3 : depuis la clef usb (package **linux-sources-2.6.24.x** disponible sur les supports d'ubuntu).
- Pourquoi cette version ? -> elle est supportée par RTAI : important de travailler sur des sources du noyau et de RTAI qui concernent la même version.

- Nous allons donc travailler sur les sources du noyau 2.6.24 (celui de Hardy Heron) :

- décompresser le sources dans **/usr/src**
- creer un lien symbolique vers les sources decompressées: **/usr/src/linux -> /usr/src/linux-sources-2.6.24.x**

RTAI



Décompresser l'archive de rtai-3.7.1.tar.bz2 dans /usr/src/

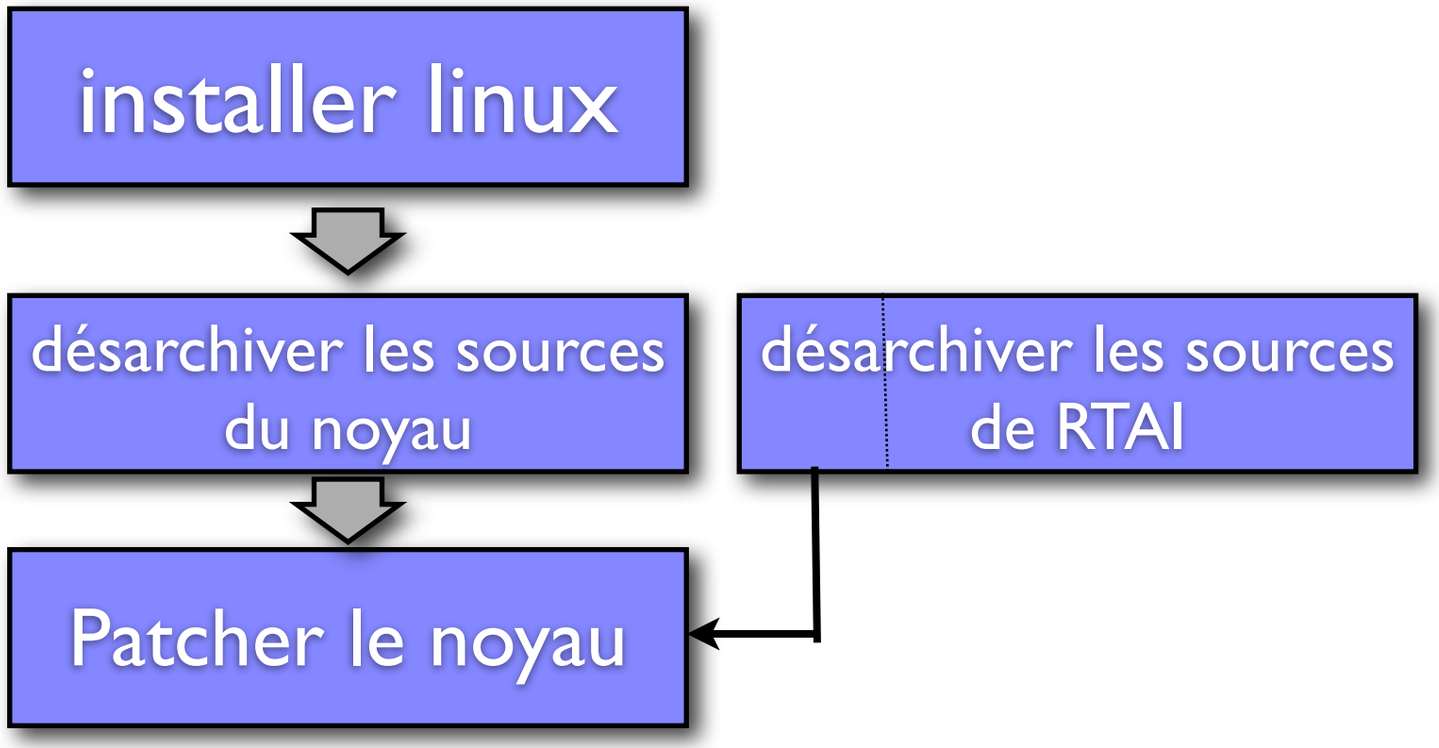
Identifier le Patch a appliquer dans :

rtai_dir/base/arch/<specific_arch>/patches

Appliquer le Patch au code source du noyau

RTAI

These RT Linux systems are **patches** to the basic Linux kernel source code.



Appliquer le Patch au code source du noyau

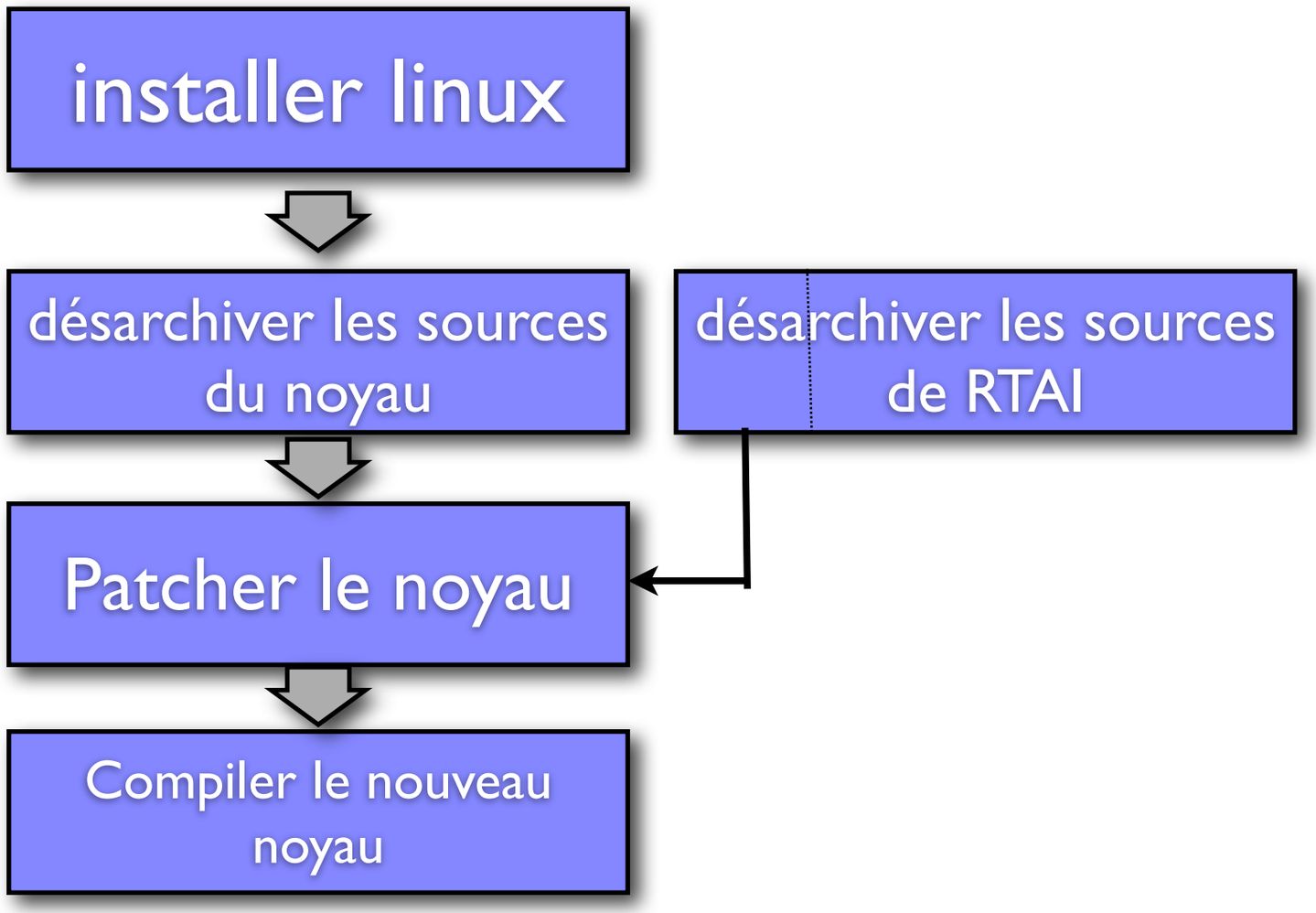
```
cd $linux_src_dir
 patch -p1 -b < rtai_dir/base/arch/<specific arch>/patches/hal-linux-2.6.24-xxx.patch
```

RTAI

http://doc.ubuntu-fr.org/tutoriel/comment_compiler_un_kernel_sur_mesure

Importer le fichier config
du noyau actuel.

cd /usr/src/linux
sudo cp /boot/config_xxx_yyy .config
sudo make oldconfig



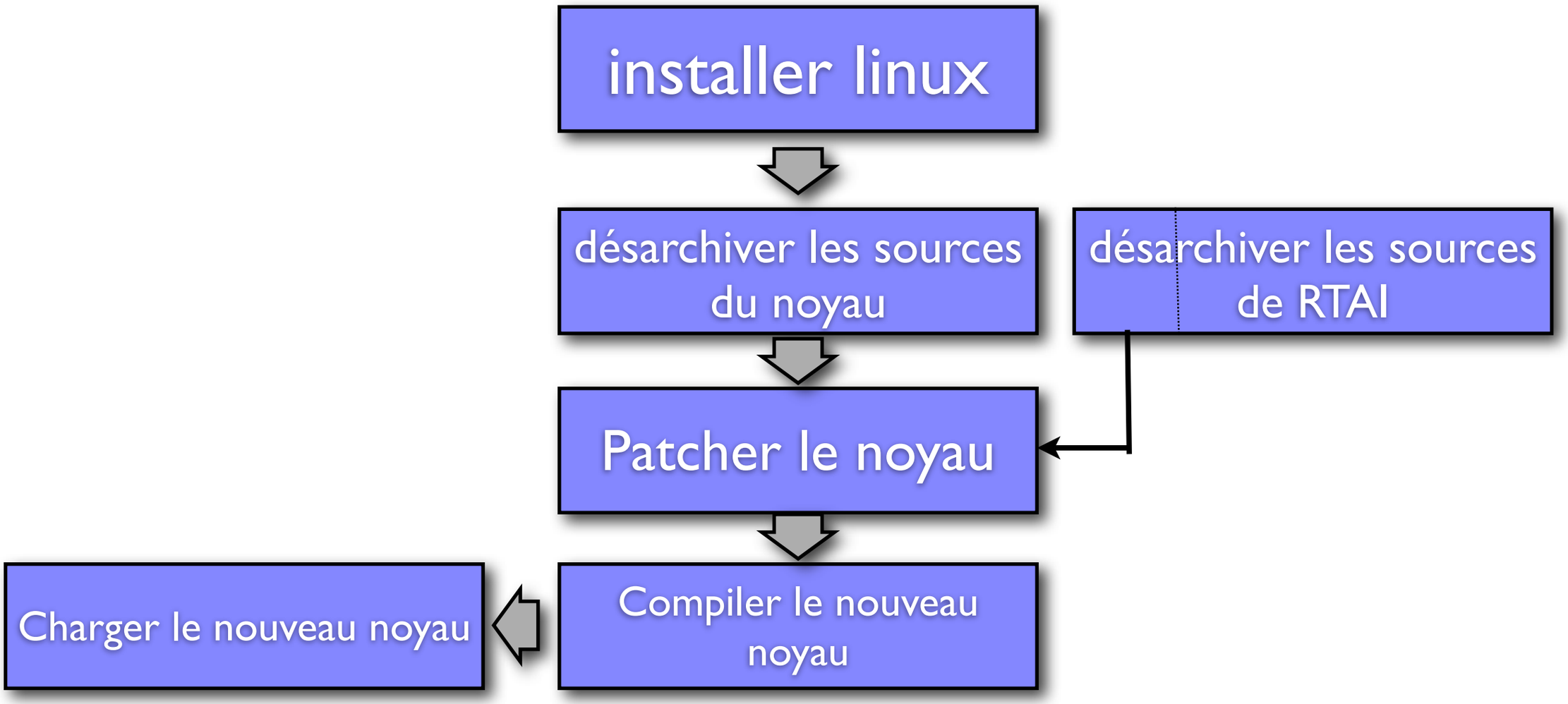

```
$sudo apt-get install libncurses5-dev
```

$ sudo make menuconfig
$ n'oubliez pas d'activer les options de compilation pour rtai (cf manuel d'installation : kernel version et pipeline interuption)

```
$ sudo make-kpkg clean
$ sudo make-kpkg --initrd --revision=386monNoyau kernel_image kernel_headers
modules_image
```

RTAI

These RT Linux systems are **patches** to the basic Linux kernel source code.

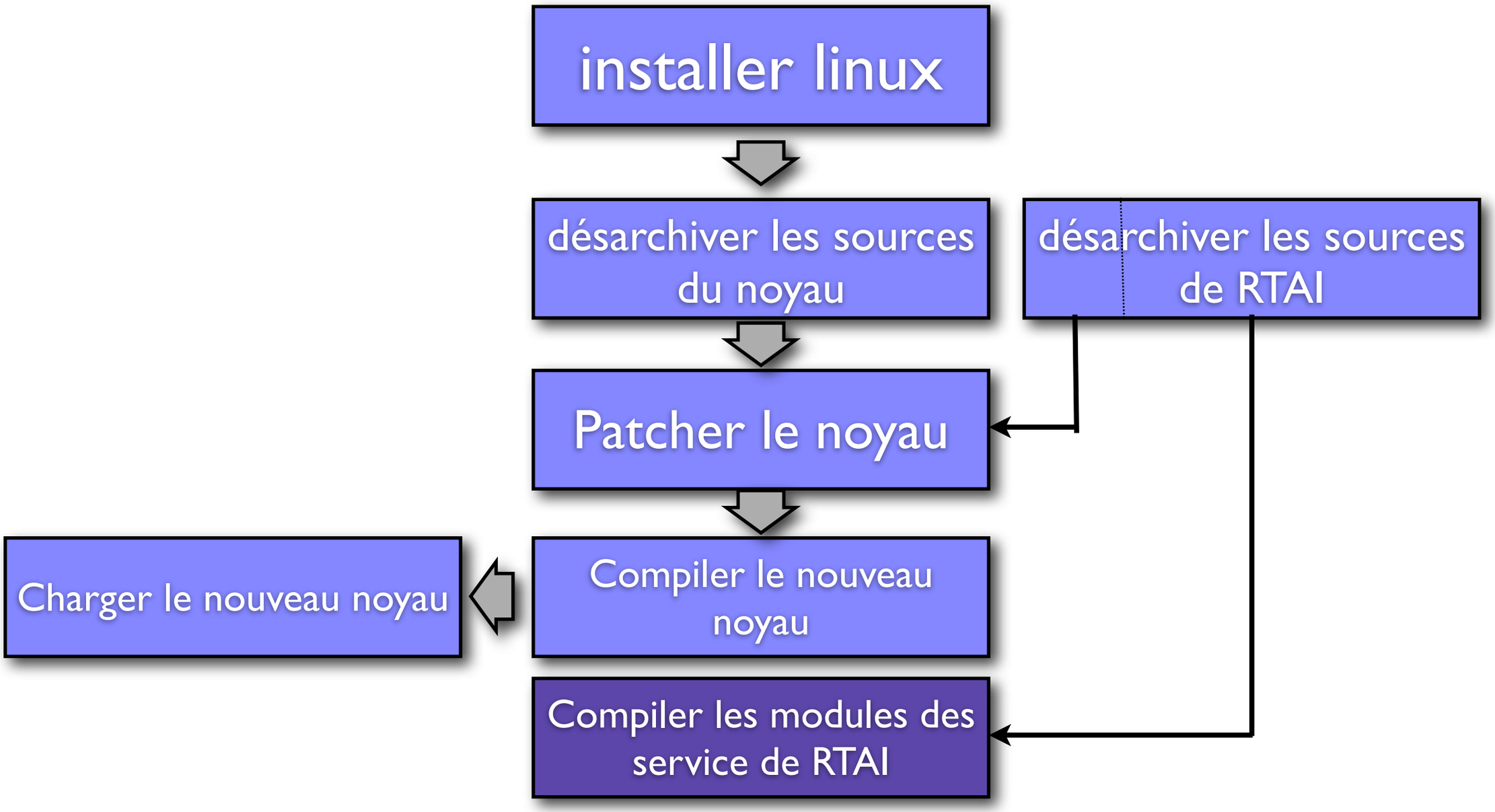



```
cd .. && dpkg -i linux*.deb
redémarrer
```

RTAI

Compiler les modules RTAI pour pouvoir les charger comme services du noyau (dormants)

RTAI


installer linux
désarchiver les sources du noyau
désarchiver les sources de RTAI
Patcher le noyau
Compiler le nouveau noyau
Charger le nouveau noyau
Compiler les modules des service de RTAI

RTAI

$ cd /usr/src/rtai-3.7.1
$ sudo make menuconfig

la compilation va générer l'ensemble des outils de RTAI dans /usr/realtime

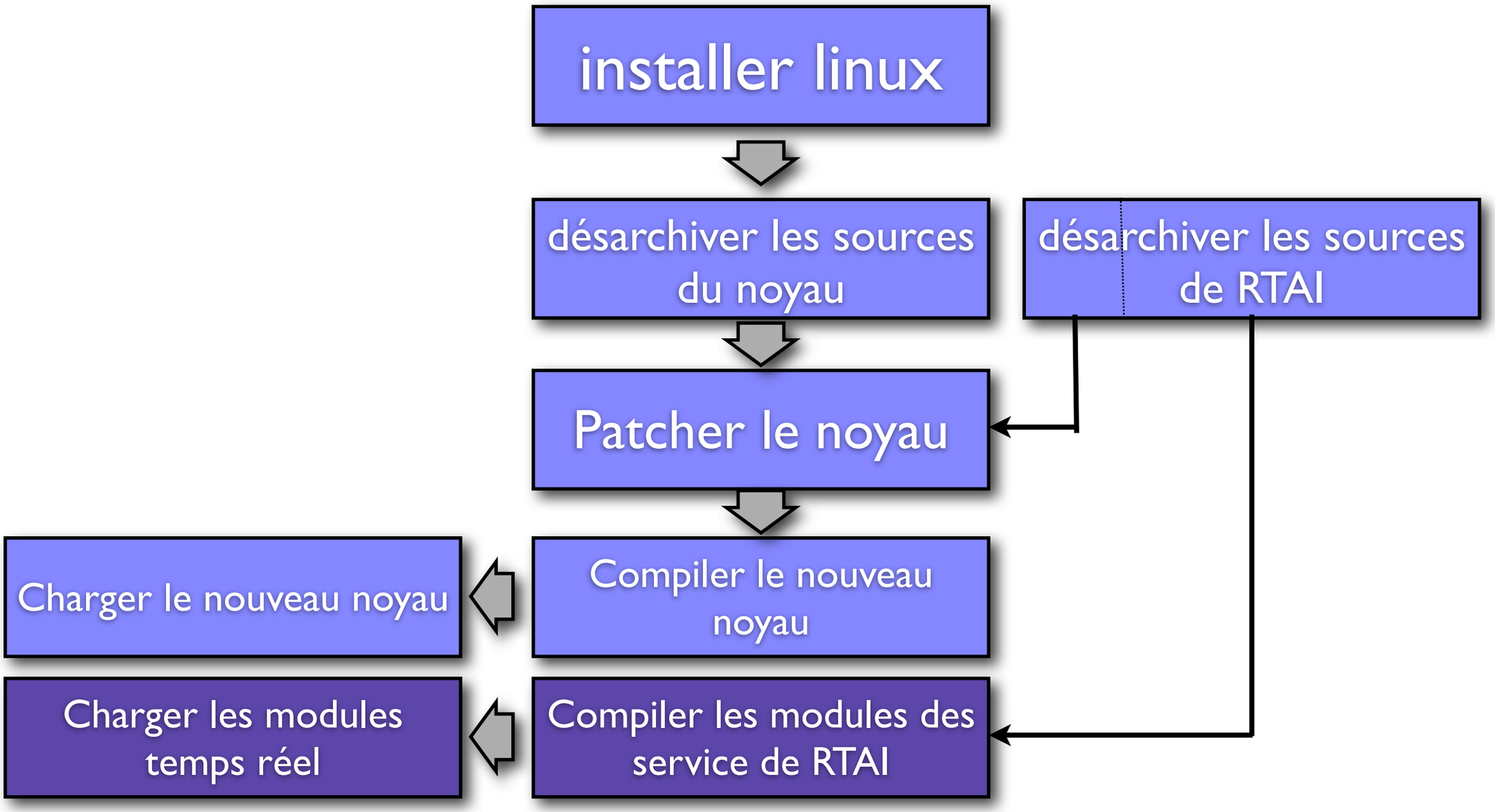
vérifiez la présence des modules propres a RTAI dans /usr/realtime/modules

-> rtai_hal.ko
->rtai_fifos.ko
etc...

vérifiez le présence des utilitaires dans :
/usr/realtime/bin

et rajouter ce dernier au PATH

RTAI


installer linux
désarchiver les sources du noyau
désarchiver les sources de RTAI
Patcher le noyau
Compiler le nouveau noyau
Charger le nouveau noyau
Compiler les modules des service de RTAI
Charger les modules temps réel

RTAI

Gestion des modules :

lsmod : liste les modules chargés

insmod xxxxx.ko : charge le module xxxxx

rmmod xxxxxx : décharge le module xxxxxx

tester le chargement des modules (dormants)

$ insmod rtai_hal.ko
$ insmod rtai_ksched.ko
$insmod rtai_fifos.ko

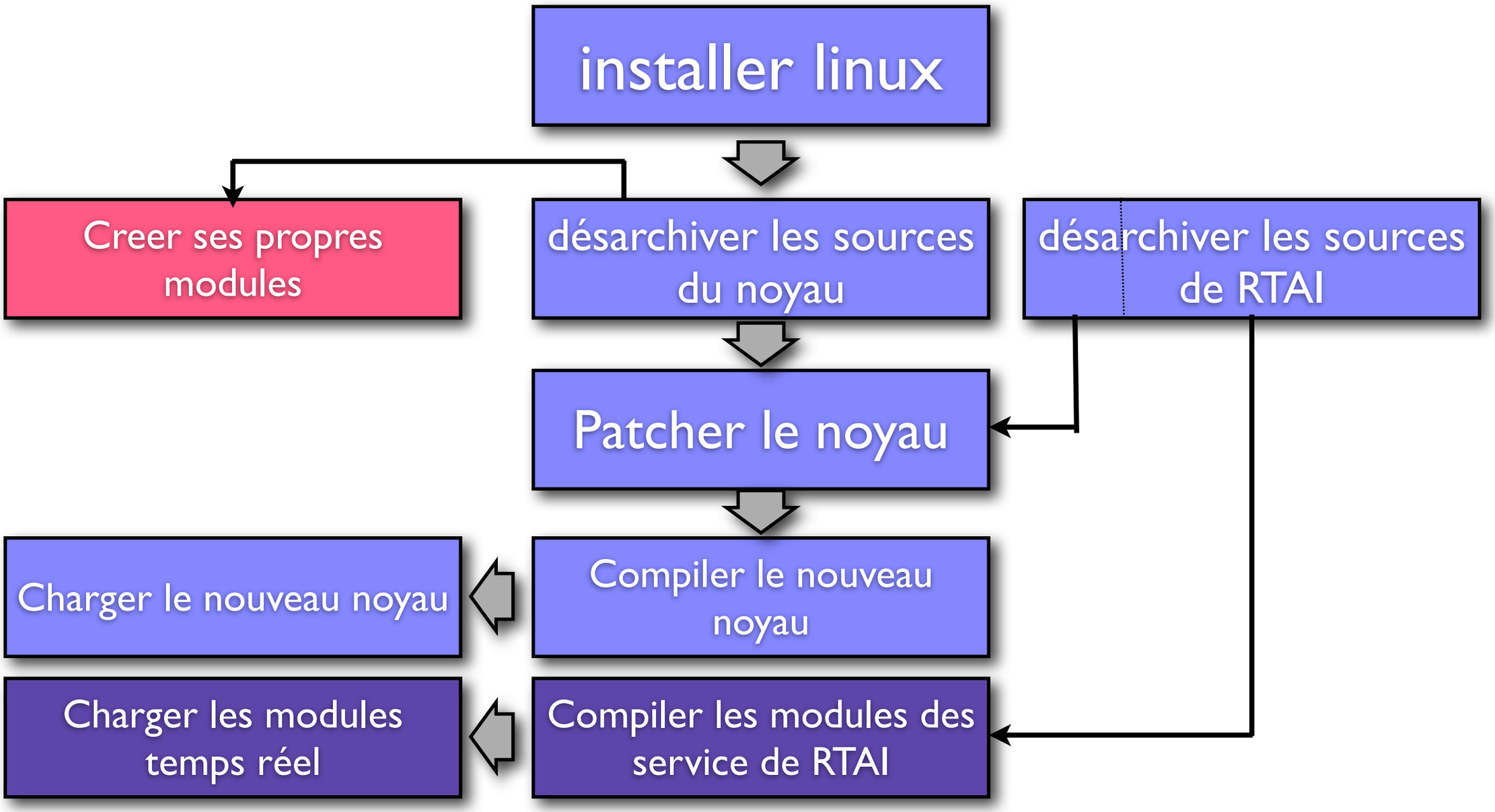
verifier les message du noyau : dmesg

RTAI

Developpement d’applications :

Creer un module du noyau standard

RTAI


installer linux
Creer ses propres modules
désarchiver les sources du noyau
désarchiver les sources de RTAI
Patcher le noyau
Charger le nouveau noyau
Compiler le nouveau noyau
Charger les modules temps réel
Compiler les modules des service de RTAI

RTAI

```c
#define __KERNEL__
#define MODULE
#include <linux/kernel.h>
#include <linux/module.h>



int init_module(void){

    printk("hello world\n");
  return 0;
}


void cleanup_module(void){


    printk("goodbye\n");
  return;
}
```

RTAI

```
obj-m := modt.o
all :
   make -C /lib/modules/2.6.24.3/build M=$(PWD) modules
clean :
   make -C /lib/modules/2.6.24.3/build M=$(PWD) clean
```

Un module est crée : xxxxx.ko
charger le module avec insmod, lister les modules avec lsmod décharger le module avec rmmod
observer les message du noyau avec dmesg

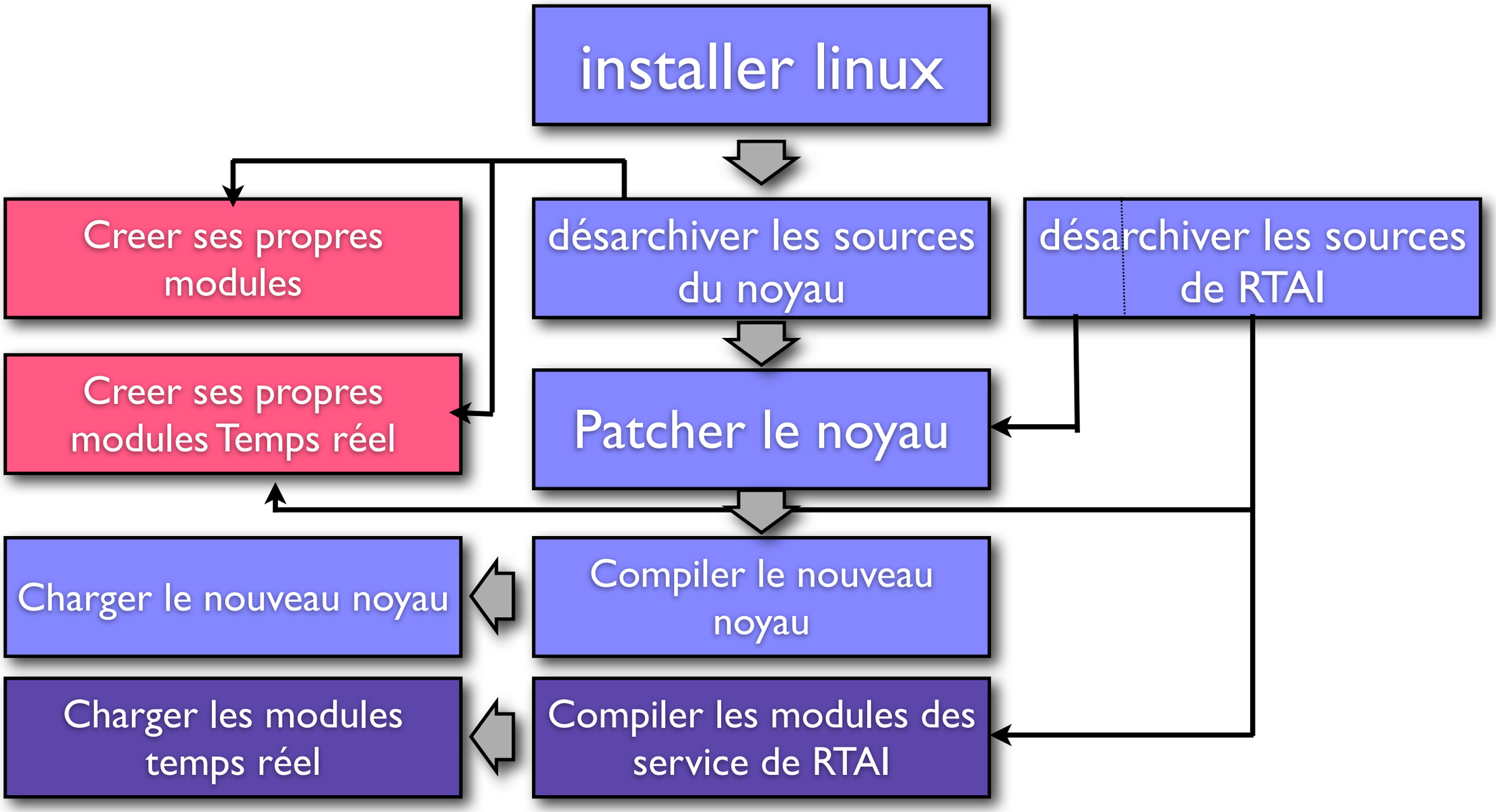
RTAI

Developpement d’applications :

Creer un module du noyau temps Réel

RTAI

These RT Linux systems are **patches** to the basic Linux kernel source code.



RTAI

Controle d’une tâche périodique en temps réel :

faire bipper le haut parleur à intervalles régulières (périodiquement):

Nous allons donc créer une tache de RTAI, qui sera gérée périodiquement par les fonctions temps réel de l’interface.

RTAI

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#define SOUND_PORT 0x061
#define SOUND_MASK 0x02


void sound_fct(){

  unsigned char sound_byte;
  unsigned char toggle = 0;

  while (1){

    sound_byte = inb(SOUND_PORT);
    if (toggle){
      sound_byte = sound_byte | SOUND_MASK;
    }
    else {
      sound_byte = sound_byte & ~SOUND_MASK;
    }
    outb (sound_byte,SOUND_PORT);
    toggle = !toggle;



  }
  return ;
}
```

bascule la valeur du bit 2
du haut parleur (adresse 61hex)

bip !

pas bip

C’est la tache (pour l’instant c’est une fonction) que nous voulons rendre périodique

RTAI

notre fonction sera liée à une tache :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;
```

```
void sound_fct(){

 ......
}
```

includes concernant le module

includes pour l'appel des fonctions temps réel

Déclaration de la tâche temps réel qui va encapsuler notre fonction

Période souhaitée en ns

RTAI

mise en place du module :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;
```

```
void sound_fct(){

 ......
}
```

```
int init_module(void){




}

void cleanup_module(void){


 }
```

RTAI

L'essentiel :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;

void sound_fct(){

 ......
}

int init_module(void){

rt_set_periodic_mode();


}


void cleanup_module(void){

 rt_task_delete(&sound_task);

 }
```

basculement en mode temps réel périodique

RTAI

L'essentiel :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;
```

```
void sound_fct(){

 ......
}
```

```
int init_module(void){

rt_set_periodic_mode();
rt_task_init(&sound_task,sound_fct,0,1024,RT_LOWEST_PRIORITY,0,0); /*initialisation*/



}
```

```
void cleanup_module(void){

 rt_task_delete(&sound_task);

 }
```

Initialisation de la tâche :

- pointeur de la tâche concernée
- fonction associée
- donnée non-utilisée
- taille de la pile
- Priorité
- calcul en virgule flottante
- fonction de gestion de signaux

après cette ligne la tâche est suspendue

RTAI

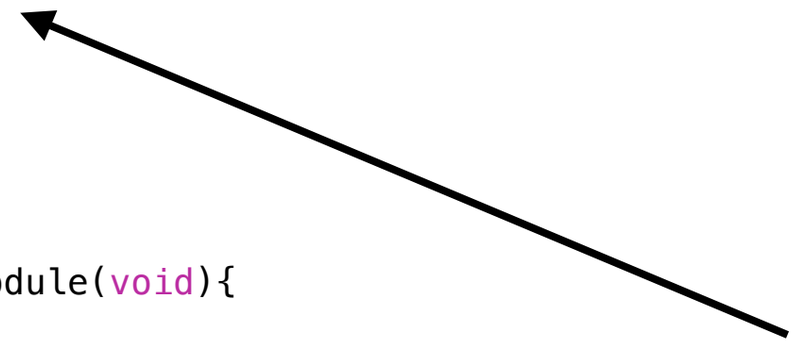
L'essentiel :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;
```

```
void sound_fct(){

 ......
}
```

```
int init_module(void){

rt_set_periodic_mode();
rt_task_init(&sound_task,sound_fct,0,1024,RT_LOWEST_PRIORITY,0,0);
rt_task_make_periodic(&sound_task, rt_get_time()+ sound_period_count,sound_period_count);


}
```

```
void cleanup_module(void){

 rt_task_delete(&sound_task);

 }
```



Réveille la tâche àprès sound_period_count puis s'executera tous les sound_period count tics d'horloge

RTAI

L'essentiel :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;

void sound_fct(){

 ......
}

int init_module(void){

rt_set_periodic_mode();
rt_task_init(&sound_task,sound_fct,0,1024,RT_LOWEST_PRIORITY,0,0);
rt_task_make_periodic(&sound_task, rt_get_time()+ sound_period_count,sound_period_count);


}



void cleanup_module(void){

 rt_task_delete(&sound_task);

 }
```

efface la tâche

RTAI

L'essentiel :

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <asm/io.h>

#include <rtai.h>
#include <rtai_sched.h>
static RT_TASK sound_task;
static RTIME sound_period_ns = 1000000;

void sound_fct(){
   while (1){
            ....
            rt_task_wait_period();
     }
}

int init_module(void){

rt_set_periodic_mode();
rt_task_init(&sound_task,sound_fct,0,1024,RT_LOWEST_PRIORITY,0,0);
rt_task_make_periodic(&sound_task, rt_get_time()+ sound_period_count,sound_period_count);


}



void cleanup_module(void){

 rt_task_delete(&sound_task);
 stop_rt_timer();
 }
```

Boucle infinie

Traitement que l'on veut faire

Attente du prochain reveil

A l'arret du module, on retrouve la résolution initale (et la fluidité) de linux

RTAI

Il reste a configurer les timers :

```
int init_module(void){


 RTIME sound_period_count;
  RTIME timer_period_count;




  rt_set_periodic_mode();

  sound_period_count = nano2count(sound_period_ns);
  timer_period_count =  start_rt_timer(sound_period_count);
  printk("periodic_task : rquested %d counts, got %d counts\n",(int) sound_period_count, (int)
timer_period_count); /*période obtenue auprès du timer*/



...

}
```

Periode souhaitée (tics)

Periode accordée a la quelle tournera le timer

Conversion ns->tics

Le timer expirera a chaque periode timer_period_count
(timer non reprogrammé a la fin de chaque cycle)
i.e Résolution des cycles de notre ordonnancement

RTAI

Code complet : sound_fct :

```
void sound_fct(){

  unsigned char sound_byte;
  unsigned char toggle = 0;

  while (1){

    sound_byte = inb(SOUND_PORT);
    if (toggle){
      sound_byte = sound_byte | SOUND_MASK;
    }
    else {
      sound_byte = sound_byte & ~SOUND_MASK;
    }
    outb (sound_byte,SOUND_PORT);
    toggle = !toggle;

    rt_task_wait_period(); /* periode defnie dans la structure task*/

  }
  return ;
}
```

RTAI

Code complet : init_module :

```
int init_module(void){
  RTIME sound_period_count;
  RTIME timer_period_count;
  int retval;

  printk("inserting sound module\n");


  rt_set_periodic_mode();
  sound_period_count = nano2count(sound_period_ns); /*conversion de la période (ns) en tics d'horloge*/
  timer_period_count =  start_rt_timer(sound_period_count); /*requette d'une période formulée en tics d'horloge*/
  printk("periodic_task : rquested %d counts, got %d counts\n",(int) sound_period_count, (int) timer_period_count); /
*période obtenue auprès du timer*/

  /*============test ici de requette de la période==================*/

  retval = rt_task_init(&sound_task,sound_fct,0,1024,RT_LOWEST_PRIORITY,0,0);

  if (retval !=0){
    if (-EINVAL == retval) printk("task already in use\n");
    else if (-ENOMEM == retval) printk ("could not allocate stack\n");
    else printk("error initializing task\n");
    return 0;
  }

  rt_task_make_periodic(&sound_task, rt_get_time()+ sound_period_count,sound_period_count);

}
```

RTAI

Code complet : clean_up module :

```
void cleanup_module(void){
  rt_task_delete(&sound_task);

  outb(inb(SOUND_PORT) & ~SOUND_MASK, SOUND_PORT); /*on eteint le son*/
}
```

RTAI

Makefile :

```
obj-m := test_sound.o
KDIR  := /lib/modules/2.6.24.3/build
EXTRA_CFLAGS := -I/usr/realtime/include -I/usr/include
PWD := $(shell pwd)

default :
   make -C $(KDIR) SUBDIRS=$(PWD) modules


clean :
   make -C $(KDIR) SUBDIRS=$(PWD) clean
```

RTAI

execution :

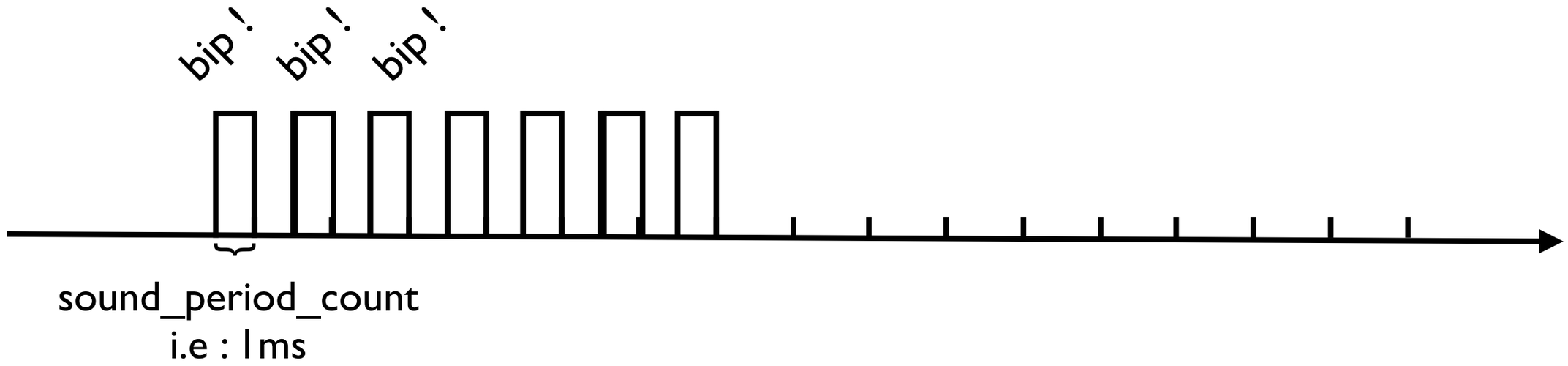
```
sudo insmod rtai_hal.ko
sudo insmod rtksched.ko
sudo insmod test_sound.ko
```

verifiez le chargement (lsmod)
verifiez les message (dmesg)

alternative : utiliser /usr/realtime/bin/rtai-load

RTAI

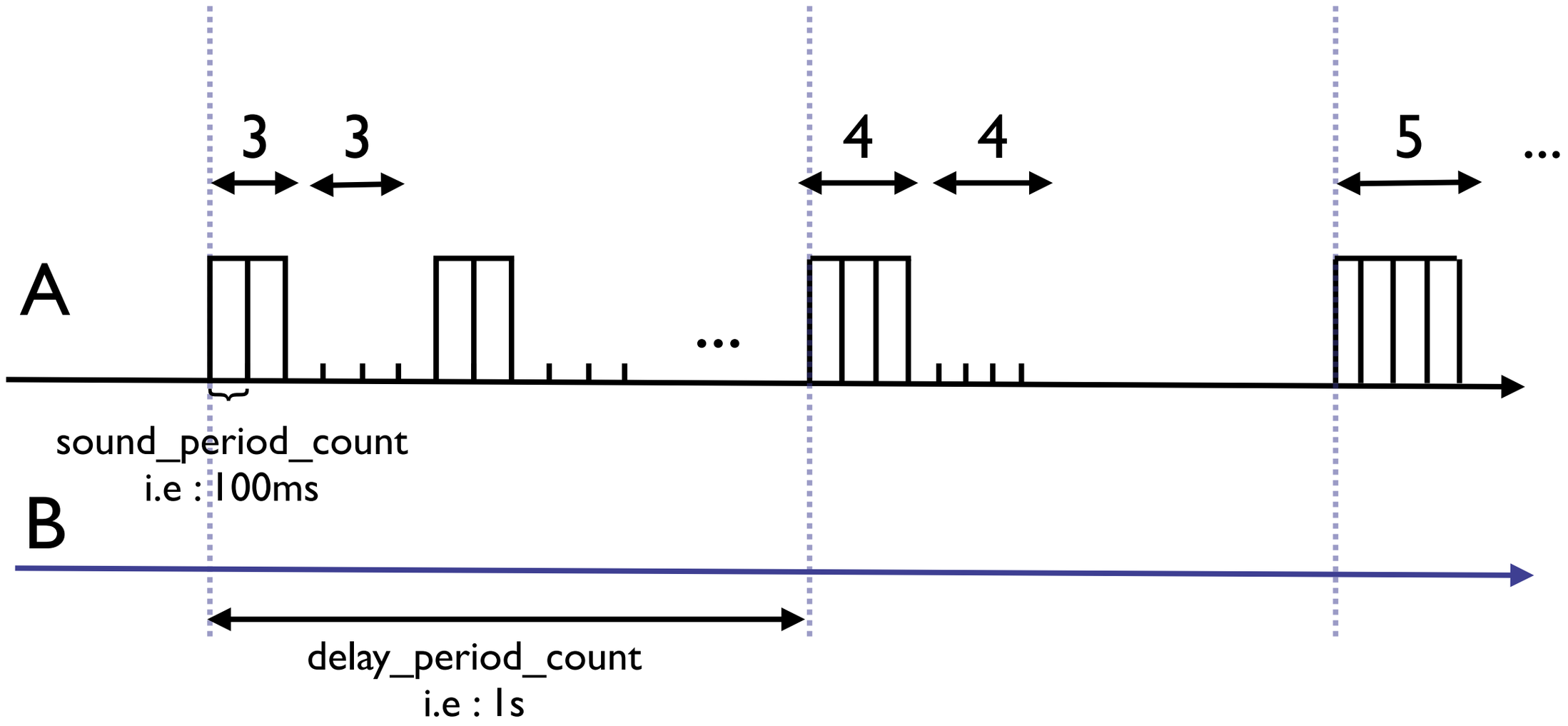
Résultat attendu :



Testez en faisant varier sound_period_ns

RTAI

Exercice : utiliser deux taches périodiques en utilisant le principe vu précédement
Tâche A : fréquence 10Hz :produit une onde “rectangulaire”
Tâche B: fréquence 1Hz : augmente la longueur du rectangle de 1 à chaque reveil



RTAI

Echange d'information avec des taches linux : FIFO

Kernel

Création d'une FIFO

```
#include <rtai_fifos.h>

#define FIFO 0

rtf_create(FIFO,8000);

rtf_put(FIFO,&cpt,sizeof(cpt));



rtf_destroy(FIFO);
```

Linux

/dev/rtf0

```
fifo = open ("/dev/rtf0",O_RDONLY)

read (fifo,&counter,sizeof(counter));
```

RTAI

```c
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/io.h>
#include <math.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>

#define TICK_PERIOD 1000000
#define TASK_PRIO 1
#define STACK_SIZE 10000
#define FIFO 0

static RT_TASK mytask;

static void fun(int t ){
printk("filling fifo\n");
  int cpt = 0;
  float sin_value = 42.;
  while (1) {
    sin_value = 42.;
    rtf_put(FIFO,&cpt,sizeof(cpt));
    rtf_put(FIFO,&sin_value,sizeof(sin_value));
    cpt ++;
    rt_task_wait_period();
    }
}

int init_module(void){
  long tick_period;
  printk("inserting\n");
      rt_task_init(&mytask,fun,1,STACK_SIZE,TASK_PRIO,1,0);
    rtf_create(FIFO,8000);
    tick_period = start_rt_timer(nano2count(TICK_PERIOD));
    rt_task_make_periodic(&mytask, rt_get_time()+tick_period, tick_period);
  return 0;
}

void cleanup_module(void){

    stop_rt_timer();
    rtf_destroy(FIFO);
    rt_task_delete(&mytask);
    printk("removing module \n");

  return;
}
```

RTAI

Echange d'information avec des taches linux : FIFO

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
```

```c
static int end;
static void endme (){
  end = 1;
}


int main (){

  int fifo, counter;
  float sin_value;
  if ((fifo = open ("/dev/rtf0",O_RDONLY))< 0) {
    fprintf(stderr,"error opening /dev/rtf0\n");
    exit (1);
  }
  printf("fifo ouverte\n");
  signal (SIGINT, endme);
  while (!end) {
    printf("lecture...counter\n");
    read (fifo,&counter,sizeof(counter));
    printf("lecture...sin\n");
    read (fifo,&sin_value,sizeof(sin_value));
    fprintf(stderr,"=> %i %f \n",counter,sin_value);

  }
      return 0;
}
```

RTAI

FIFO : ecriture de Linux vers le module RTAI :

Kernel

```
num_read = rtf_get(0, &buffer_in, sizeof(buffer_in));
```

non bloquante, retourne 0 si rien a lire

Linux

```
num_written = write(write_descriptor, &buffer_out,
sizeof(buffer_out));
```

Pour éviter de faire une scrutation de la FIFO, on peut associer un “Handler” :

- One optional feature of FIFOs is the "handler," a function on the real-time side that is called whenever a FIFO is read or written. To install a handler, call '**rtf_create_handler()**' with the FIFO number and the name of the handler function.
- A handler is often used in conjunction with 'rtf_get()' to process data acquired asynchronously from a Linux process. The installed handler calls 'rtf_get()' when data is present. Because the handler is only executed when there is activity on the FIFO, polling is not necessary.

RTAI

FIFO : ecriture de Linux vers le module RTAI :

Kernel

```
num_read = rtf_get(0, &buffer_in, sizeof(buffer_in));
```

non bloquante, retourne 0 si rien a lire

Linux

```
num_written = write(write_descriptor, &buffer_out,
sizeof(buffer_out));
```

Pour éviter de faire une scrutation de la FIFO, on peut associer un “Handler” :

- One optional feature of FIFOs is the "handler," a function on the real-time side that is called whenever a FIFO is read or written. To install a handler, call '**rtf_create_handler()**' with the FIFO number and the name of the handler function.
- A handler is often used in conjunction with 'rtf_get()' to process data acquired asynchronously from a Linux process. The installed handler calls 'rtf_get()' when data is present. Because the handler is only executed when there is activity on the FIFO, polling is not necessary.

RTAI

Exemple de handler

```
static int fifo_handler(unsigned int fifo)
{
  ....

  int num;

  /*  Lecture complète de la FIFO */
  do {
    num = rtf_get(RTF_COMMAND_NUM, &command, sizeof(command));
  } while (num != 0);

    /* traitement des données ici */

  ....

  ....

  return 0;
}
```

appel du handler, dans init_module, après la création et l'initialisation de la FIFO

```
rtf_create_handler(RTF_COMMAND_NUM, fifo_handler);
```

RTAI

```
static int fifo_handler(unsigned int fifo)
{
  ....

  int num;

  /*  Lecture complète de la FIFO */
  do {
    num = rtf_get(RTF_COMMAND_NUM, &command, sizeof(command));
  } while (num != 0);

    /* traitement des données ici */

  ....

  ....

  return 0;
}
```

appel du handler, dans init_module, après la création et l'initialisation de la FIFO

```
rtf_create_handler(RTF_COMMAND_NUM, fifo_handler);
```

RTAI

Mémoire partagée :
Utiliser une structure commune ou l'information est stockée.
La structure n’est plus une file, mais les information sont écrasées a chaque écriture
Les champs de la structure peuvent être accédés (lu/écris) indépendamment

Plusieurs tâches peuvent donc agir indépendamment sur la zone de mémoire partagée

Attention a la consistance des données.

Une tâche RTAI peut interrompre un process Linux
Pas l’inverse

2 cas :

- Si RTAI est le lecteur : interruption de l’écriture en cours par Linux -> données inconsistantes (données nouvelles + donnée anciennes)
- Si RTAI est le rédacteur : interruption de la lecture de Linux (données anciennes + données nouvelles)

A VOUS DE GERER LA CONSISTANCE

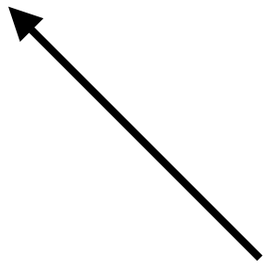
RTAI

Mémoire partagée : Création

```
#include "rtai_shm.h"
```

Kernel

```
struct mystruct * ptr;
ptr = rtai_kmalloc(101, sizeof(struct mystruct));
```

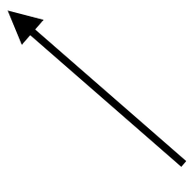


Mémoire partagée avec la clé 101
Doit être créee du coté temps réel avant d'être utilisée dans Linux

```
#include <rtai_shm.h>
```

Linux

```
struct mystruct * ptr;
ptr = rtai_malloc(101, sizeof(struct mystruct));
```



Plus une référence qu'une allocation (effectuée coté temps réel)

La clé 101, doit être la même des deux coté
La valeur de ptr, elle est différente.

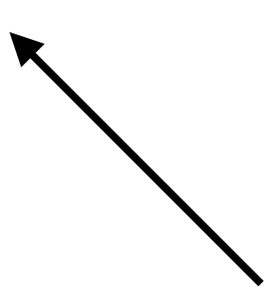
RTAI

Mémoire partagée : Suppression

```
#include "rtai_shm.h"
```

Kernel

```
rtai_kfree(101);
```

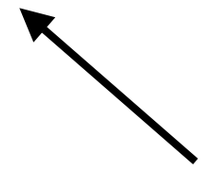


Libération de la mémoire
en second

```
#include <rtai_shm.h>
```

Linux

```
rtai_free(101, ptr);
```



désassociation
(en premier)

!!!! ORDRE INVERSE DE LA CREATION

RTAI

Gestion de la consistance : exemple :

- il existe mutexes et semaphores pour gerer la consistance entre taches Linux
- il existe mutexes et semaphores pour gerer la constistance entre taches RTAI

- Besoin d'outils entre Linux et RTAI

mauvais exemple :

les deux taches peuvent modifier token

```
int token = 0;

if (token == 0) { /* no one has it */
   token = 1; /* now I have it */
     /* modify shared data here */
     token = 0; /* now I give it up */
   } else {
     /* I didn't get it; I'll try again */
   }
```

échec

RTAI

Gestion de la consistance : exemple :

Algorithme de **Dekker's et Peterson** pour 2 taches

les deux taches modifient une variable commune, mais surtout : deux variables d'intentions différentes

```
int favored_process = 0;
int p0_wants_to_enter = 0;
int p1_wants_to_enter = 0;



      #include "rtai_shm.h"
/* how p0 gets the resource */
p0_wants_to_enter = 1;
favored_process = 1;
while (p1_wants_to_enter && favored_process ==
1) /* spinlock */ ;
/* operate on shared resource here */
p0_wants_to_enter = 0; /* give up the resource */



/* how p1 gets the resource */
p1_wants_to_enter = 1;
favored_process = 0;
while (p0_wants_to_enter && favored_process ==
0) /* spinlock */ ;
/* operate on shared resource here */
p1_wants_to_enter = 0; /* give up the resource */
```

RTAI

TEST :

Ecrire une tache périodique RTAI qui :

- incremente une variable heartbeat
- rempli un tableau partagé avec la **meme** valeur (par exemple heartbeat)

Ecrire une tache linux qui lit périodiquement le tableau

- taux d'erreur ?

tester avec et sans protection

RTAI

Sémaphores :

gérer l'ordonnancement et l'accès aux sections critiques de plusieurs tâches RTAI
similaire à POSIX

Initialisation :

```
SEM sem; /* here's the semaphore data structure */
rt_sem_init(&sem, 1); /* 1 means it's initially available */
```

Invocation du sémaphore :

```
rt_sem_wait(&sem); /* blocked waiting... */
/* we returned, and now have the semaphore */
```

Rends le sémaphore :

```
rt_sem_signal(&sem); /* will always return immediately */
```