Les OS pour l'IoT

Jérôme Ermont

Toulouse INP/ENSEEIHT/Sciences du Numérique

Plan de la présentation

Un OS pour l'IoT ?

- Un objet connecté est composé de :
  - MCU
  - Capteurs, actionneurs
  - Circuit de communication
- Du matériel hétérogène :
  - Différent type de MCU (ARM, TI, ...)
  - Différents piles de communication (BLE, TSCH, ...)
  - Différents capteurs

$\rightarrow$ Redéfinir le code pour chaque élément !

Un OS

- Portage du code
- Abstraction du matériel
- Gère la multiprogrammation

Les contraintes

Contraintes matérielles

- Processeur simple, non prévu pour le calcul
- Peu de mémoire
- Nécessité d'une faible consommation d'énergie

Un OS pour l'IoT doit

- Avoir peu de calcul à effectuer, seulement permettre la collecte des données
- Être compact = taille du code réduite
- Ne s'exécuter qu'en cas de besoin
  - Event driven kernel = réveil à l'arrivée d'un événement
- Posséder des mécanismes de sauvegarde d'énergie

Différentes solutions existent

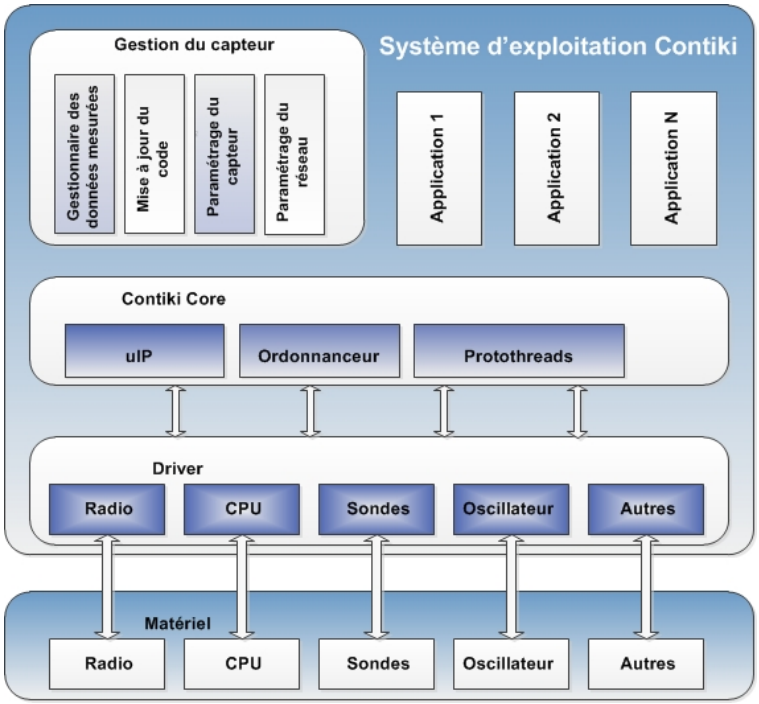
- Quelques unes :
  - tinyOS, OpenWSN, MS Windows IoT, ARM mbedOS, Intel Zephyr, Huawei LiteOS, ...
- Pour ce cours, 3 technologies parmi les plus utilisées
  - Contiki-ng : Evolution de Contiki, Event-driven kernel
  - RIOT : Micro-noyau, modulaire
  - FreeRTOS : OS temps réel

Plan de la présentation

Contiki-NG

- 2017 : Evolution (nouvelle branche) de Contiki (2002)
- Amélioration de la pile de communication IPv6
- Ajout de plateformes IoT modernes
- Amélioration de la structure interne
- Distribué sous licence BSD

Architecture



Source : Wikipedia

- Pilotes pour les périphériques = abstraction du matériel
- Multi programmation événementielle : Protothreads
- Empreinte mémoire : $\sim$100kO de ROM et $\sim$ 10kO de RAM pour l'OS

Un exemple : Helloworld

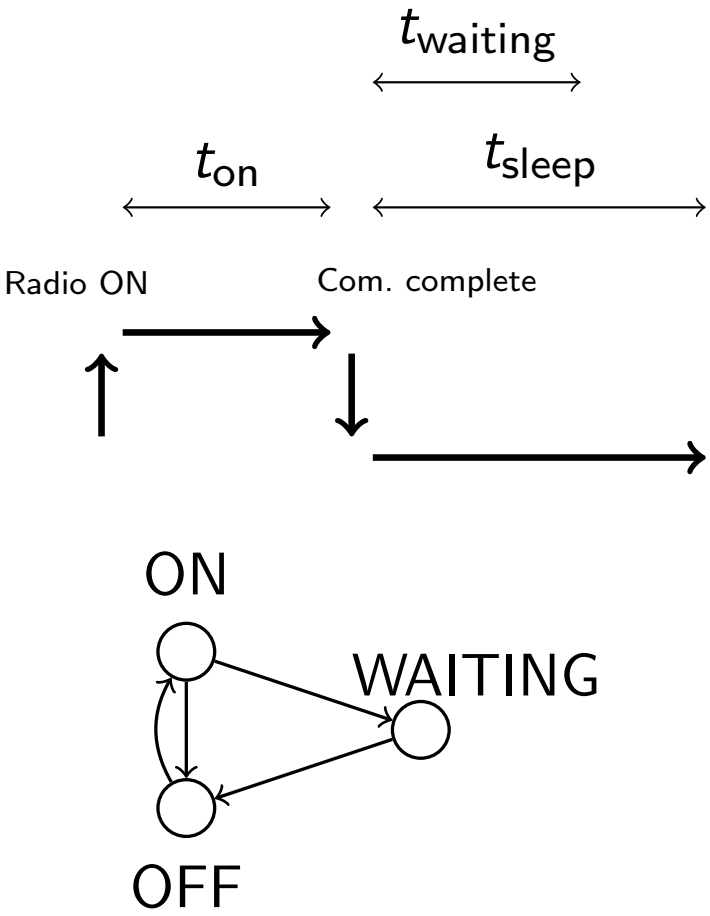
```
// Declaration d'un processus
PROCESS(hello_world_process, "Hello world");

// Lancement du processus au chargement du systeme
AUTOSTART_PROCESSES(&hello_world_process);

// Code du processus
PROCESS_THREAD(hello_world_process, ev, data) {
        PROCESS_BEGIN();
        printf("Hello, world!\n");
        while (1) {
                PROCESS_WAIT_EVENT(); // Attente d'un
        }
        PROCESS_END();
}
```

Processus Contiki et Protothreads

- Le noyau est basé sur les événements
  - Une seule pile, moins de mémoire
  - Lancement des processus lorsque quelque chose se produit = signal de capteurs, expiration d'un timer
  - Le lancement ne doit pas être bloqué
- Le comportement d'un processus est implanté via le concept de protothread
  - Le code du processus est un protothread
- Protothreads = exécution séquentielle dans un processus

Protothread vs machine à état

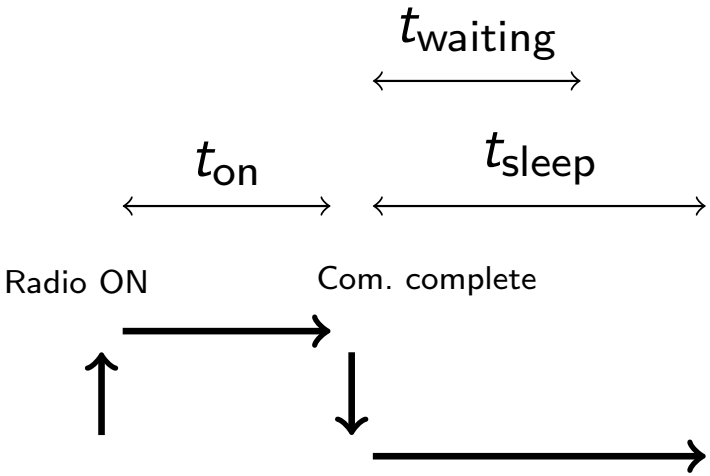
- Fonctionnement par machine à états




```
typedef enum {ON, WAITING, OFF} t_state;
t_state state= OFF;

void gestion_evenement() {
        if (state == ON) {
                if (expired(timer))(
                        if (!comm_complete()){
                                state= WAITING;
                        } else {
                                radio_off();
                                state= OFF;
                        }
                } else if (state == WAITING) {
                        if (comm_complete()) {
                                radio_off();
                                state= OFF;
                        }
                } else if (state == OFF) {
                        if (expired_timer)) {
                                radio_on();
                                state= ON;
                        }
                }
        }
```

Prototheread vs machine à état

- Fonctionnement avec Prototheads




```
int protothread(struct pt *pt) {
PT_BEGIN(pt);
while (1) {
        radio_on();
        PT_WAIT_UNTIL(pt, expired(timer));
        if (!comm_complete()) {
                PT_WAIT_UNTIL(pt, comm_complete()
                    || expired(timer_wait));
        }
        radio_off();
        PT_WAIT_UNTIL(pt, expired(timer));
}
PT_END(pt);
}
```

- Code plus compact

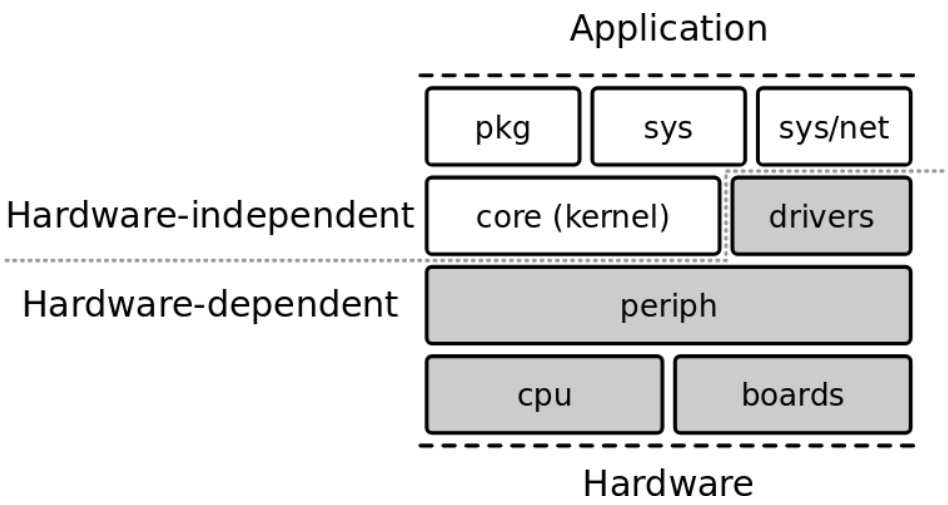
Timer

4 types de timers

- struct timer :
  - Ne fait qu'informer de l'expiration de timer
- struct etimer :
  - Envoie un événement à l'expiration du timer
- struct ctimer :
  - Appelle une fonction à l'expiration du timer
- struct rtimer :
  - Timer temps réel, appelle une fonction à la date précisée

Plan de la présentation

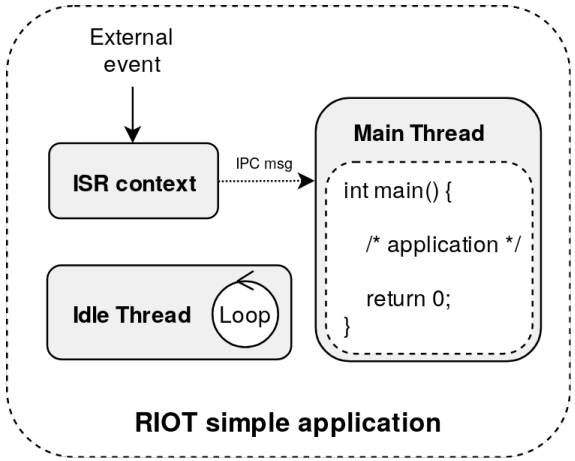
RIOT



- Approche μ-noyau : flexible et modulaire
- Multi-plateforme : AVR, ARM, MIPS32, MSP430, PIC32, RISC-V, x86, ...
- Empreinte mémoire : $\sim$ 1.5ko RAM, $\sim$ 5ko ROM

Caractéristiques

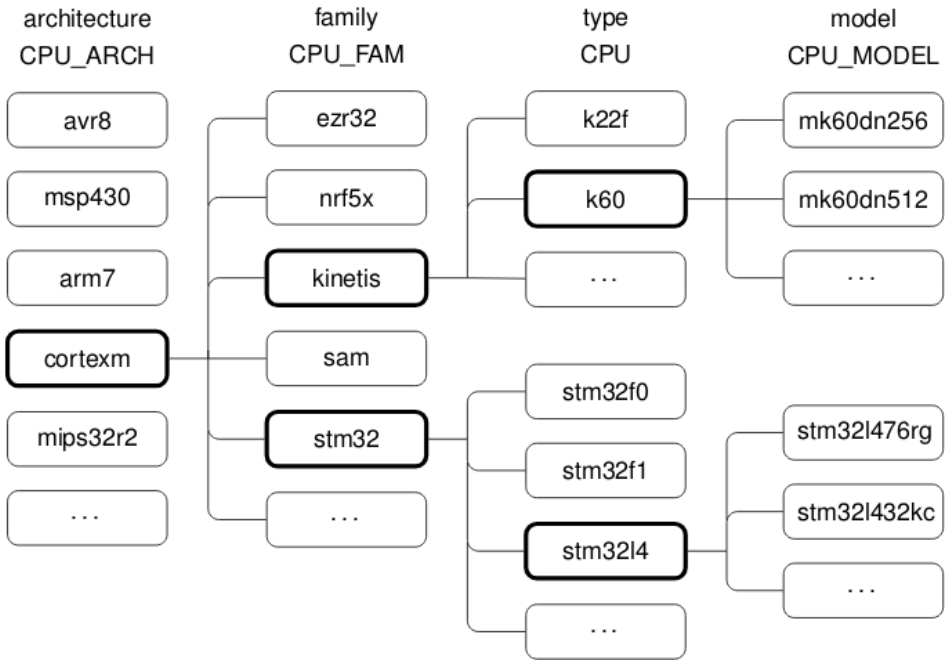
- Mutil-tâches
- Noyau temps réel à priorité fixe
- Mécanismes de communications inter-tâches (IPC)
- Mécanismes de synchronisation : partage de ressources (Mutex/Sémaphore)
- Différents threads :
  - thread principal : *main thread*, code principal de l'application
  - thread d'attente : *idle thread*, code exécuté lorsque les autres threads sont terminés, gère la gestion d'énergie



Source : Alexandre Abadie, INRIA

Couche d'Abstraction Matérielle (HAL)

- Niveau CPU :
  - éléments de base commun à plusieurs cartes
  - spécificités des MCU
  - classé en architecture (ARM, MIPS, ...), famille (STM32, SAM, ...), type, modèle
  - évite la redondance de code



E. Baccelli et al. (2018) "RIOT : an Open Source Operating System for Low-end Embedded Devices in the IoT", IEEE Internet of Things Journal. PP. 1-1. 10.1109/JIOT.2018.2815038

Couche d'Abstraction Matérielle (HAL)

- Niveau Carte :
  - Configuration des périphériques (UART, SPI, I2C, ...)
  - Configuration des outils
  - Macros pour utiliser les LEDs, boutons, capteurs, ...

- l'API Périphérique :
  - Interface commune pour différents types d'architectures/familles/types
  - Portabilité du code
  - Périphérique : timer, uart, spi, i2c, pwm, adc, ...
  - Ajout d'une fonctionnalité au système : `FEATURES_REQUIRED = +periph_<nom du périphérique>` dans `Makefile` + `#include <periph/nom du périphérique>` dans le `.c`

Exemple de périphérique : GPIO

- GPIO : General Purpose I/O
- API à inclure : `periph_gpio`
- Ajout des fonctions au code l'application : `#include <periph/gpio.h>`
- Initialisation du port I/O : `gpio_init()`, valeurs possibles : INPUT, INPUT avec pull-down, INPUT avec pull-up, OUTPUT, etc
- Mise à un : `gpio_set()` et à zéro : `gpio_clear()`
- Utilisation des IT : `gpio_init_int()`
- Doc : `https ://doc.riot-os.org/gpio_8h.html`

Exemple de code d'application pour RIOT

```c
#include <stdio.h>

#include "shell.h"
#include "thread.h"

static char stack[THREAD_STACKSIZE_MAIN];

static void *thread_handler(void *arg)
{
    (void)arg;
    puts("Hello from thread!");
    return NULL;
}

int main(void)
{
    thread_create(stack, sizeof(stack), THREAD_PRIORITY_MAIN - 1,
                  0, thread_handler, NULL, "new thread");

    char line_buf[SHELL_DEFAULT_BUFSIZE];
    shell_run(NULL, line_buf, SHELL_DEFAULT_BUFSIZE);
    return 0;
}
```

Plan de la présentation

FreeRTOS

- FreeRTOS est un système temps réel utilisé pour les processeurs embarqués
- Utilise un micro-noyau et est sous licence GPL modifiée
- Simple et léger
- Fonctions pour : multi-threads, mutex, semaphores, timers
- Mode tick-less : arrêt des interruptions liées à l'horloge $\rightarrow$ réduit la consommation d'énergie

Gestion des tâches

- Création de tâche : xTaskCreate
- Destruction : vTaskDelete
- Attente d'un délai : vTaskDelay

Sémaphore

- Sémaphore binaire : vSemaphoreCreateBinary
- Sémaphore mutex : xSemaphoreCreateMutex
- Prendre le sémaphore : xSemaphoreTake
- Libérer le sémaphore : xSemaphoreGive