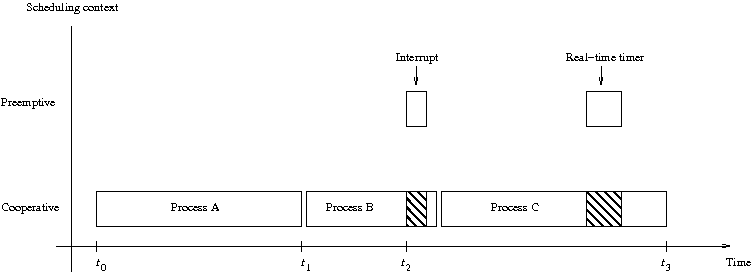
Détails sur l’organisation et les sous-systèmes de Contiki

# I / Processus



Le code dans Contiki peut être exécuté soit en exécution coopérative soit en exécution préemptive. Les processus sont toujours exécutés en coopératifs et les uns après les autres. Lorsqu’il y a un déclenchement d’une minuterie ou un événement externe (exécution préemptive) cela stop le processus en cours et il ne reprend que lorsque cette « interrupt » ou ce « real-time timer » est terminé.

## Description d’un processus sur Contiki :

Une image contenant texte

Description générée automatiquement

## Protothreads

Le système Contiki n’utilise pas des threads basiques puisqu’ils surchargent la mémoire (pour que chacun puisse accéder au même bloc de donnée. Ici, on utilisera les protothreads. Ils permettent d’exécuter plusieurs activités lorsque notre code attend que quelque chose se produise.

**Contiki implémente sa propre version des protothreads :**

* PROCESS\_BEGIN() ; //déclare le début du protothread d’un processus
* PROCESS\_END() ; //déclare la fin du protothread d’un processus
* PROCESS\_EXIT() ; //quitte le processus
* PROCESS\_WAIT\_EVENT() ; //attend un événement
* PROCESS\_WAIT\_EVENT\_UNTIL() ; //attend un événement avec une condition
* PROCESS\_WAIT\_UNTIL() ; //attend une condition donnée
* PROCESS\_YIELD() ; //attend un événement, voir plus bas les effets secondaires
* PROCESS\_PAUSE() ; //renvoie temporairement le processus

## Evénements

**Dans Contiki, un processus est exécuté lorsqu’il reçoit un événement.**

**Evénements asynchrones :** ils ne sont pas directement délivrés au processus de réception, ils sont d’abord mis dans une file d’attente à l’intérieur du noyau. Celui-ci va les parcourir et « réveillé » le processus lié à l’événement. Si l’événement est lié à tous les processus du système alors le noyau le fait l’un après l’autre. La fonction process\_post() permet de publier l’événement dans la file d’attente.

**Evénements synchrones** : ils sont directement livrés au processus. Celui-ci ne peut pas distinguer si l’événement reçu est synchrone ou non.

**Vote :** c’est un type d’événement. La fonction process\_poll() permet d’interroger un processus. C’est avec cela qu’on fait des interruptions. Cette fonction est donc la seule que l’on peut appeler dans le mode préemptif en toute sécurité.

**Identificateurs d’événement :** chaque événement possède un identifiant, celui-ci est codé sur 8 bits. On transmet aussi ce numéro au processus qui reçoit l’événement. Ce nombre permet de savoir quel type d’événement il a reçu. Les identifiants au-dessus de 128 compris sont définissable que par le noyau. En voici quelques un :

* PROCESS\_EVENT\_INIT = 129 //envoyé lorsque le processus est créé
* PROCESS\_EVENT\_POLL = 130 //envoyé lorsque le processus est en cours d’interrogation
* PROCESS\_EVENT\_EXIT = 131 //envoyé lorsque le processus est en train d’être tué par le noyau, le processus peut choisir de nettoyer toutes les ressources allouées
* PROCESS\_EVENT\_CONTINUE = 133 //envoyé lorsque le processus attend une instruction PROCESS\_YIELD()
* PROCESS\_EVENT\_MSG = 134 //envoyé lorsque le processus reçoit un message, beaucoup utilisé par la pile IP mais peut aussi s’utiliser inter processus
* PROCESS\_EVENT\_EXITED = 135 //envoyé lorsqu’un autre processus est sur le point de se terminer, il reçoit aussi un pointeur vers l’espace alloué pour le processus bientôt terminé (ils pourront par exemple nettoyer les allocations)
* PROCESS\_EVENT\_TIMER = 136 //envoyé lorsqu’un timer a expiré

## Le planificateur de processus

Le planificateur de processus est fait pour appeler les processus lorsqu’il faut qu’ils s’exécutent. Celui-ci appelle la fonction qui implémente le thread du processus. C’est toujours le résultat d’un événement (ou d’un vote). L’événement est transmis avec son identifiant mais peut aussi transmettre un pointeur s’il doit transmettre des données (sauf pour les votes toujours NULL).

La fonction **process\_start()** démarre un processus. Cette fonction configure la structure de contrôle du processus, ajoute le processus dans la liste des actifs et appelle le code d’initialisation dans le thread. Si le processus existe déjà, elle le retourne. L’état du processus est mis sur **PROCESS\_STATE\_RUNNING** et le protothread du processus est sur **PT\_INIT().** Ensuite, il reçoit directement un **PROCESS\_EVENT\_INIT()** qui vient du processus qui a lancé **process\_start()** ce qui active la partie initialisation du code.

Un processus a 2 manières de se terminer. Soit il se tue, soit il est tué. Dans le premier cas, il utilise **PROCESS\_EXIT()** ou il atteint une instruction **PROCESS\_END().** Dans le second cas, un autre processus appelle **process\_exit()** sur notre processus. À la suite d’une mort, le noyau envoi le **PROCESS\_EVENT\_EXIT** à tous les autres processus. Il est enfin supprimé de la liste des actifs.

Démarrage automatique de processus : peut surgir lors du démarrage du système ou lorsqu’un module est chargé. Ceux qui se démarrent au démarrage du système sont dans une liste spéciale contenue dans l’un des modules utilisateurs. Lorsqu’un module est chargé, le chargeur de module va aussi consulter la liste des processus à démarrer automatiquement est va les charger.

## Conclusion

Pour exécuter une application sur Contiki, il faut quasiment toujours passer par un processus. Un processus = bloc de contrôle (informations d’exécution) + protothread (code). Protothread = thread léger spécial pour les OS à mémoire limitée. Un code s’exécute soit en coopératif (**processus**) soit en préemptif (**interruptions + process\_poll()**). Les processus communiquent entre eux avec les événements.

# II / Timers

## Les modules de minuterie Contiki

## Le module horloge

## La bibliothèque du minuteur

## La bibliothèque Stimer

## La bibliothèque Etimer

## La bibliothèque Ctimer

## La bibliothèque Rtimer

## Conclusion