Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

p.03 - Systèmes d'exploitation

Classes T-2/I-2 // 2017-2018

Daniel Gachet | HEIA-FR/TIC

p.03 | 19.03.2018

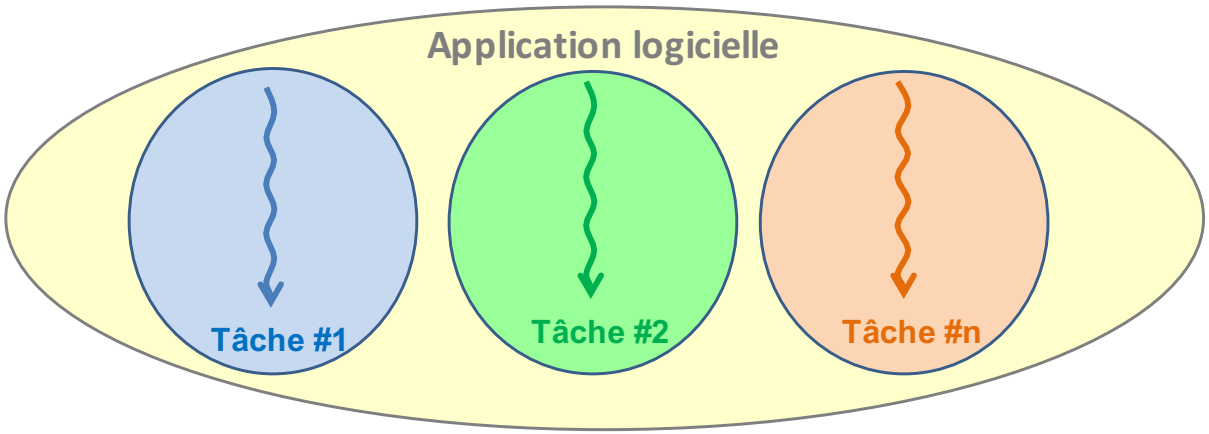
Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

Contenu

- Introduction
- Processus vs Thread
- Composantes d'un noyau

Principe

- Avec des systèmes à µP, il est courant de devoir traiter plusieurs tâches en parallèle



- Plusieurs techniques sont à disposition pour simplifier le développement de telles applications
- Quelques exemples
  - Simple Control Loop (SCL)
  - Interrupt Controlled System (ICS)
  - Multitasking (p.ex. TNKernel)

Principe (II)

- Avec des µP de nouvelle génération, l'utilisation d'un système d'exploitation multitâche *(multitasking operating system)* est devenue courante
- Quelques exemples d'OS propriétaires
  - VRTX, Nucleus, VxWorks, QNX, OS9…
- Quelques exemples d'OS libres
  - Linux, Xenomai, RTAI, FreeRTOS/SaveRTOS…

Systèmes d'exploitation multitâche

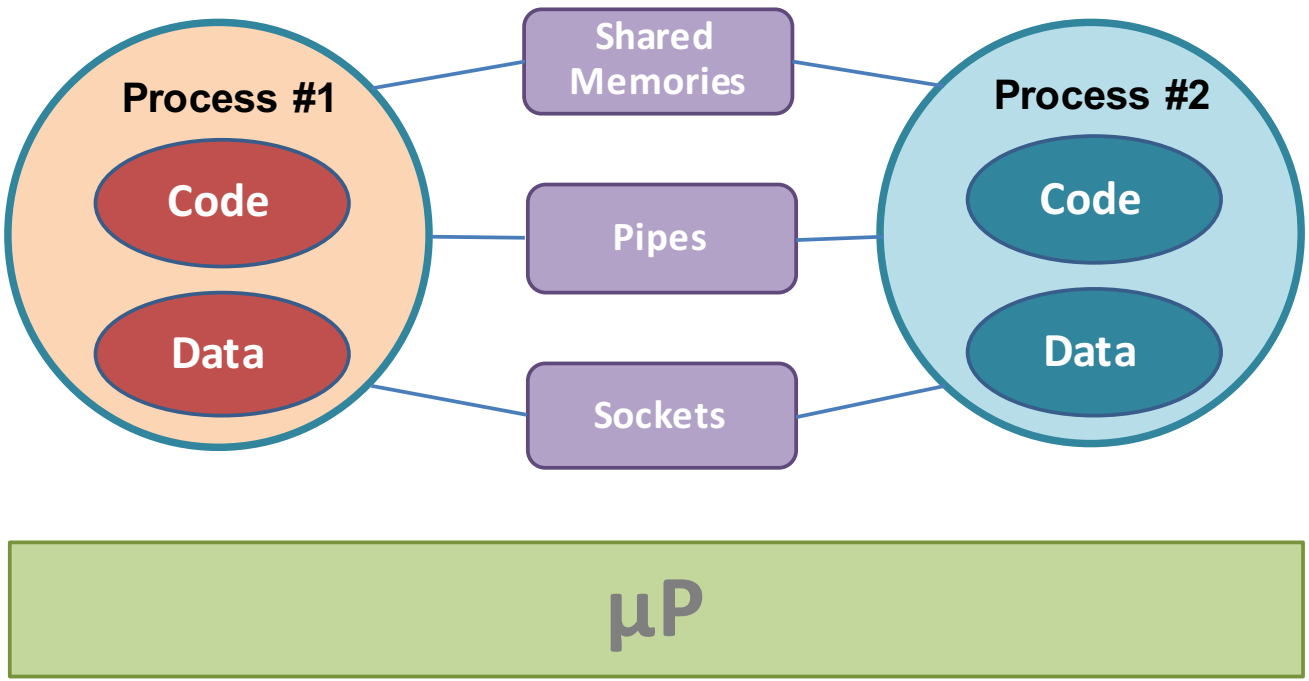
- Multitâche coopératif
  - Forme la plus simple d'un système d'exploitation multitâche
  - Chaque processus doit explicitement donner la main à une autre tâche pour s'exécuter
  - Partage inefficace des ressources processeur, grande latence sur des événements
- Multitâche préemptif
  - Forme actuelle de la plupart des systèmes d'exploitation
  - Le processeur, à l'aide d'un timer, signale au système d'exploitation de commuter le processus en cours pour un autre
  - Système robuste et très réactif
  - Chaque processus a l'illusion de disposer du processeur pour son usage exclusif

Noyaux

- Tous les systèmes d'exploitation multitâche ont en commun un noyau *(kernel)*
- Il se compose généralement de 7 éléments principaux
  - Processus, co-routines/tâches *(processes, threads/tasks)*
  - Ordonnanceur *(scheduler)*
  - Compteur périodique *(periodic timer)*
  - Mécanismes d'échange d'information *(p. ex. boîtes aux lettres)*
  - Mécanismes de synchronisation *(p. ex. sémaphores)*
  - Gestionnaire de mémoire *(heap management)*
  - Mécanisme pour le traitement des interruptions *(interrupt handler)*

Processus

- Le processus, élément de base du multitâche, est constitué de 3 éléments principaux
  - Le code
  - Les données
  - La communication

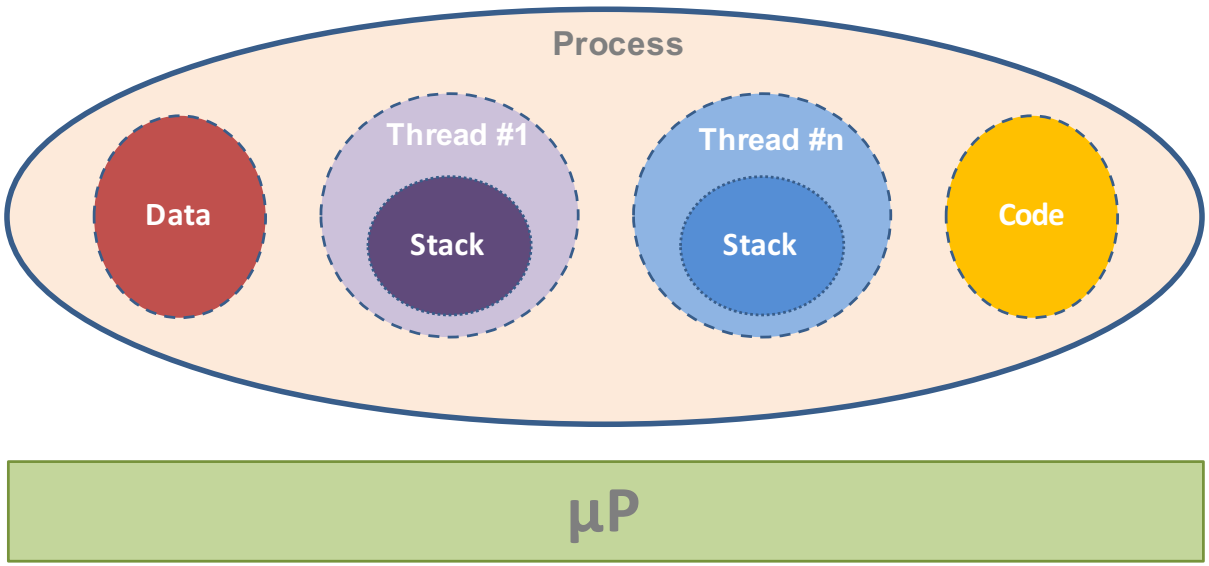


Processus (II)

- Le code
  - Algorithme permettant le traitement de la tâche pour laquelle il a été conçu
  - Le code est généralement unique pour un processus donné
  - Avec les OS modernes, il est possible de partager du code grâce aux bibliothèques partagées *(shared libraries)*
- Les données
  - Ensemble des informations à traiter
  - Les données sont toujours propres à un processus et sont contenues dans une mémoire virtuelle propre au processus *(own virtual memory)*
- La communication
  - Pour interagir, les processus disposent de divers mécanismes de communication pour l'échange d'information *(Inter Process Communication - IPC)*
  - Quelques mécanismes : shared memory, pipes, sockets,...

Thread

- Le thread, appelé aussi processus léger *(lightweight process)*, est une forme simplifiée d'un processus
- Il est constitué de 3 éléments principaux
  - Le code
  - Les données
  - La pile



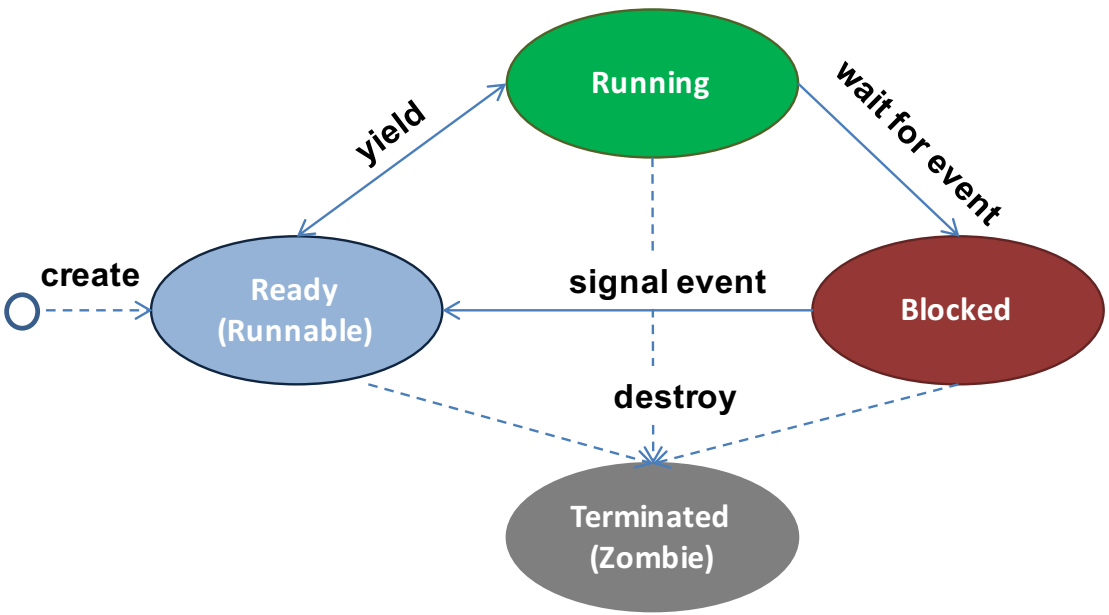
- Les threads ne peuvent exister qu'à l'intérieur d'un processus

Thread (II)

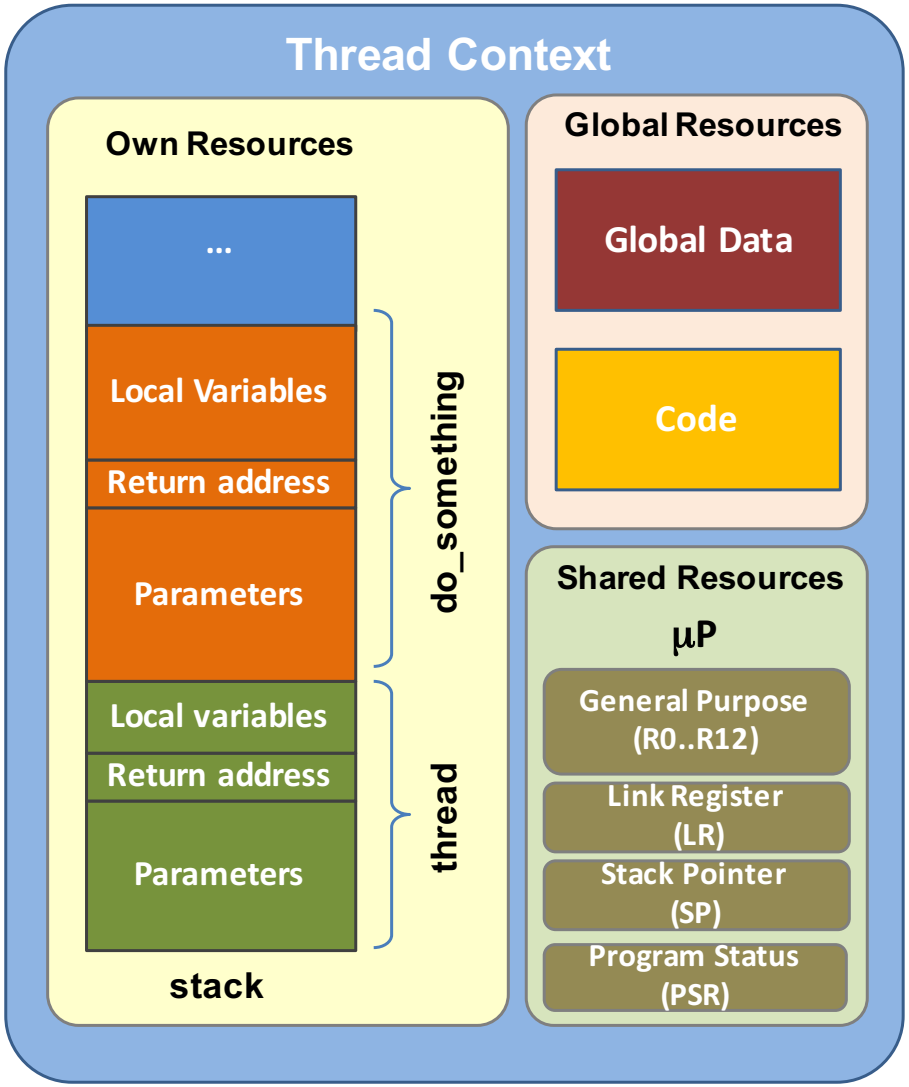
- Le code
  - Les threads permettent, comme les processus, un traitement parallèle de l'information
  - Le même code est partagé entre tous les threads du processus parent
- Les données
  - Tous les threads d'un même processus partagent la même zone mémoire *(shared memory)*
- La pile
  - Chaque thread possède sa propre pile d'appels *(own stack)* contenant les données locales

Etats d'un thread

- Durant leur fonctionnement, les threads connaissent différents états
  - Ready (Runnable) : prêt à être exécuté, en attente du processeur
  - Running : en cours d'exécution
  - Blocked : suspendu, en attente d'un événement / d'une ressource
  - Terminated (Zombie) : détruit



Contexte d'un thread



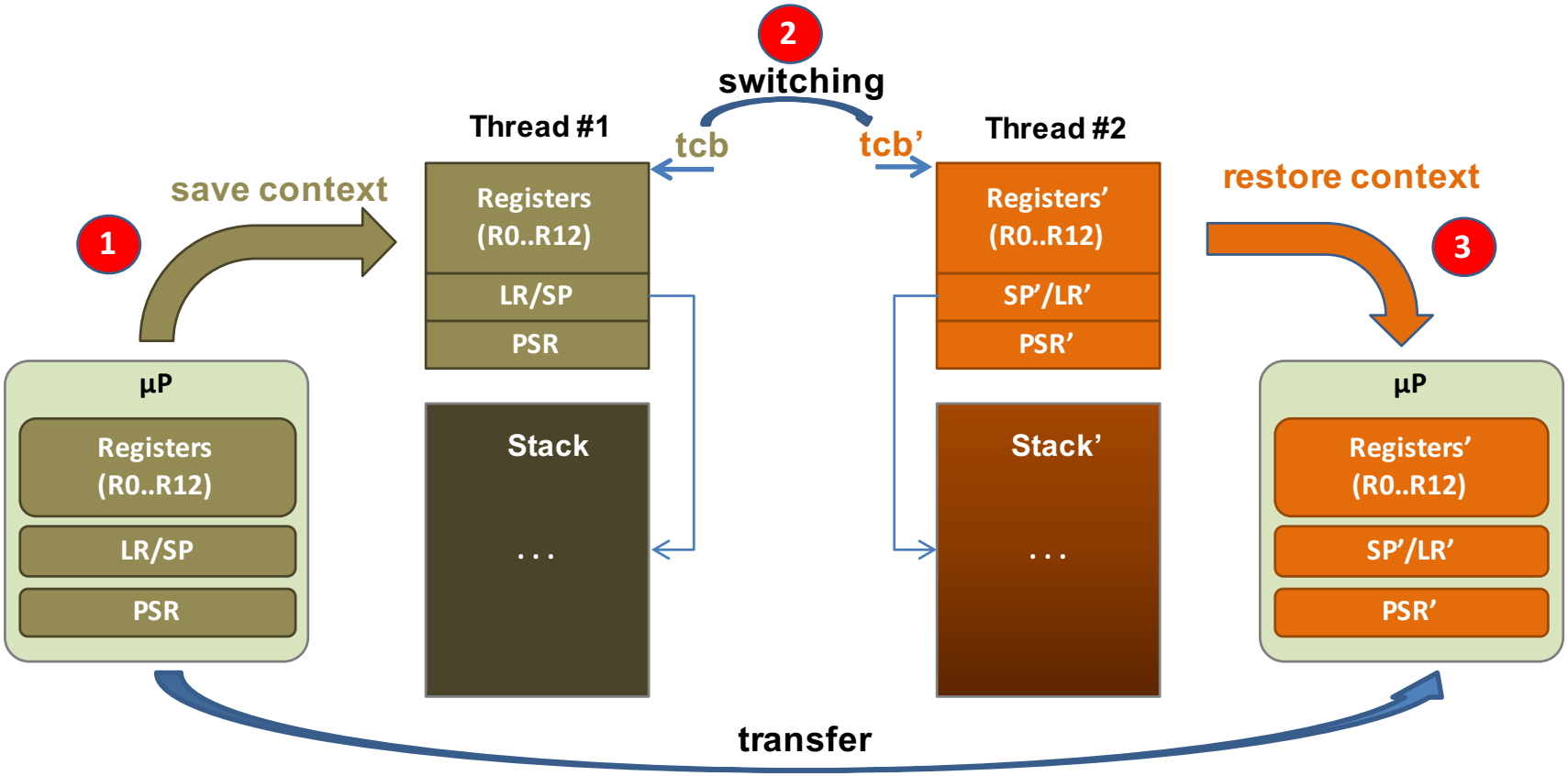
Contexte d'un thread (II)

- Du point de vue d'un langage de programmation tel que C, un thread peut être implémenté comme une sous-routine (une fonction) contenant avec une boucle infinie pour le traitement de l'information

```c
void thread ( /* parameters */ )
{
  while (1) {
     do_something ( /* parameters */ );
  }
}
```

- L'incarnation d'un thread est réalisée par l'attribution d'une pile unique, contenant l'état du thread ainsi que ses données propres

Commutation de contexte



Commutation de contexte (II)

- La commutation de contexte *(thread context switching)* s'exécute en 3 étapes
  - Sauvegarde de l'état actuel du processeur pour le thread en cours d'exécution dans une structure nommée TCB *(Thread Control Block)*
  - Commutation des TCB du thread en cours d'exécution pour le nouveau thread
  - Restauration de l'état du processeur pour le nouveau thread à partir de son TCB

Initialisation du contexte d'un thread (I)

- Concept
  - L'initialisation du contexte d'un thread doit être réalisée dans l'esprit d'une commutation de contexte ordinaire
  - Cette méthode évite d'exécuter des opérations complexes lors de son lancement/activation
- Réalisation
  - On initialise le TCB de telle manière que l'ordonnanceur appelle une routine chargée de lancer le thread avec les bons paramètres et le bon état du processeur

```
void kernel_thread_startup (void* parameter, kernel_thread_t thread)
{
  thread (parameter);
  kernel_thread_exit();
}
```

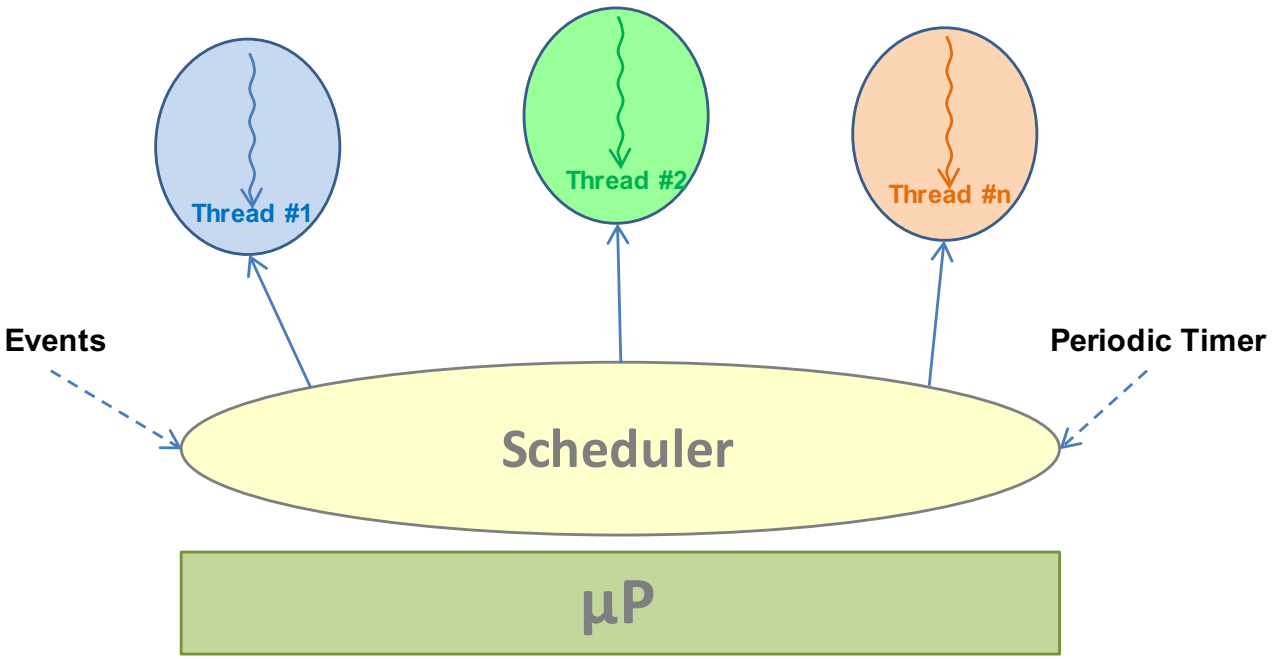
  - Ce code sera également chargé d'appeler une fonction qui terminera correctement l'exécution du thread au cas où celui-ci retournerait au programme appelant

Initialisation du contexte d'un thread (II)

- L'initialisation consiste à mettre les valeurs suivantes dans le TCB
  - R0 = paramètre du thread
  - R1 = adresse du thread (son point d'entrée)
  - R2 – R12 = valeur de défaut des registres, initialisée à 0 (zéro)
  - LR = adresse du code de démarrage du thread (startup)
  - SP = adresse sur le sommet de la pile
  - PSR = statut du processeur initialisé selon le mode de fonctionnement souhaité
    - Fanions = 0
    - IRQ/FIQ = 0
    - Mode = superviseur (svc)

Scheduler

- L'ordonnanceur *(scheduler)* est le « coeur » du système d'exploitation



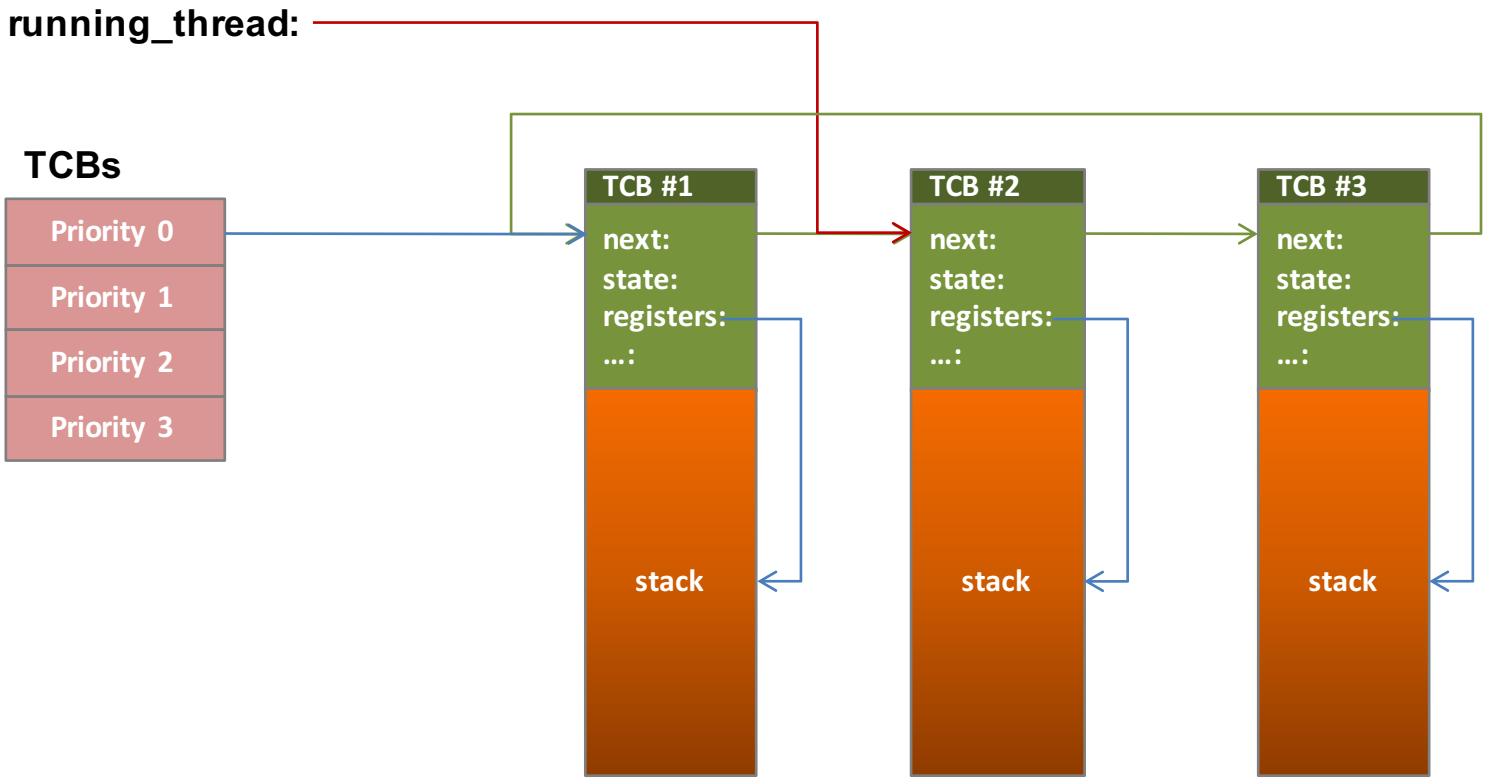
- L'ordonnanceur se charge d'attribuer le µP à un processus/thread à un temps donné

Scheduler (II)

- Il existe deux types d'ordonnanceur : coopératif et préemptif
- Cooperative Scheduler
  - Le changement de contexte est volontaire, c.à.d. le thread décide de céder la main
  - Il intervient également lorsque le thread se met en attente sur une ressource
- Preemptive Scheduler
  - Le changement de contexte intervient sur événement, soit de façon périodique via le compteur périodique, soit sur interruption matérielle
  - Il intervient également lorsque le thread se met en attente sur une ressource

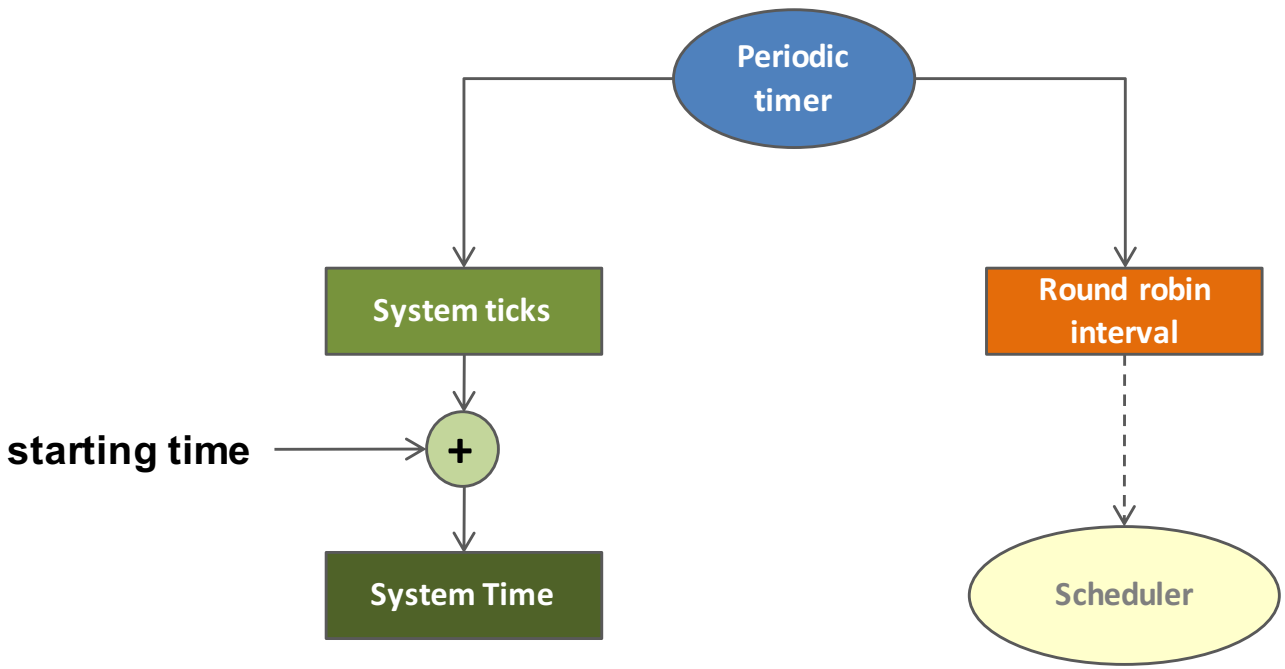
Gestion des threads

- La gestion des threads peut être simplement réalisée
  - Plusieurs niveaux de priorités
  - Listes chaînées par niveau de priorité
  - Pointeur sur le thread en cours d'exécution



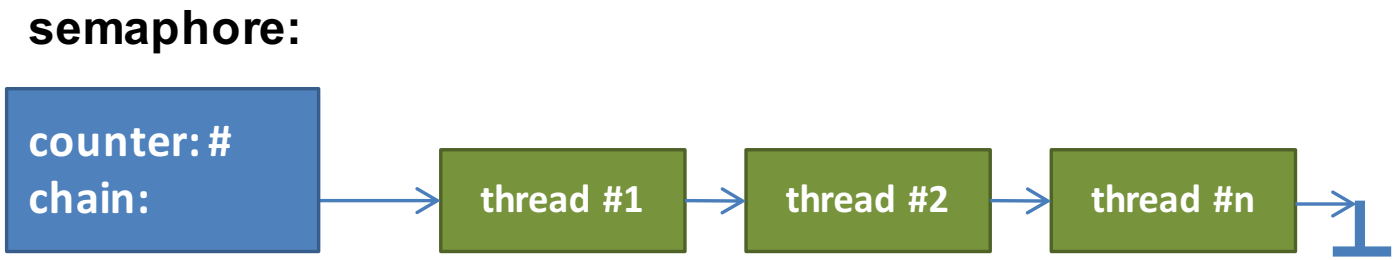
Compteur périodique

- Le compteur périodique *(periodic timer)* est un composant essentiel des systèmes d'exploitation multitâche
- Deux fonctions principales
  - Génération d'un événement vers l'ordonnanceur pour activer le changement de contexte
  - Création d'une horloge système fournissant le temps système au système d'exploitation



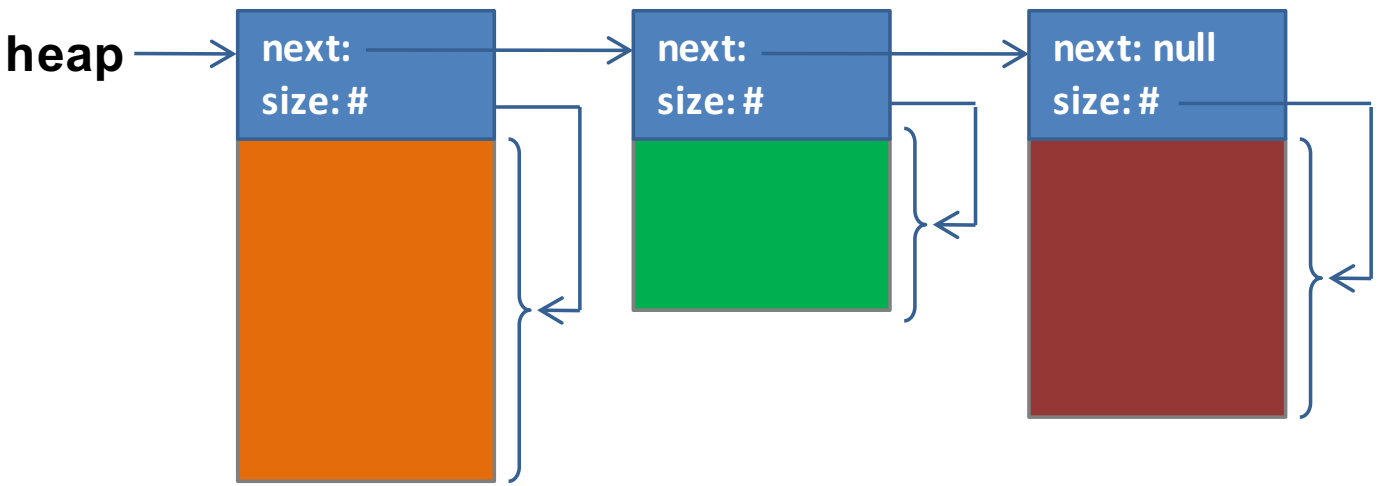
Sémaphores

- Les sémaphores fournissent un mécanisme simple, mais très efficace pour la synchronisation entre processus ou entre threads
- Ils ont été proposés par Edsger Wybe Dijkstra en 1968
- Un sémaphore est réalisé à l'aide d'une variable contenant 2 attributs
  - Compteur, si le compteur est plus grand que 0 l'accès est autorisé, sinon le processus/thread doit être mis en attente
  - Liste de processus/threads en attente



Heap Memory Management

- L'allocation dynamique de mémoire se fait depuis la mémoire tas
- La mémoire tas *(heap memory)* est une ressource globale contenant toutes les zones mémoire libres du système, c.à.d. les zones mémoires non utilisées pour le code et les variables globales
- Elle est gérée comme une liste chaînée de blocs mémoires non utilisés



Heap Memory Management (II)

- Des algorithmes plus ou moins complexes peuvent être implémentés par les systèmes d'exploitation afin d'éviter la fragmentation de cette mémoire (division de la mémoire en petits morceaux ne permettant plus l'allocation de grands blocs)
- Quelques algorithmes
  - Fixed-size-block : allocation de blocs a taille constante
  - Best fit : recherche du meilleur bloc
  - Buddy blocks : recherche du bloc par puissance de 2
- La bibliothèque « stdlib.h » fournit les méthodes « malloc/free » permettant de gérer les allocations dynamiques de mémoire

Interrupt Handler

- La gestion des interruptions est un aspect essentiel des systèmes d'exploitation multitâche
- A défaut de pouvoir implémenter le traitement complet des interruptions, ils fournissent généralement
  - Abstraction de l'architecture du processeur
  - Service de sauvegarde et de restauration du contexte
  - Primitives permettant aux applications logicielles écrites avec des langages évolués de se connecter très facilement au vecteur d'interruption
  - Mécanisme pour le traitement d'interruptions multiples/partagées