Coroutines

Les coroutines sont un concept apparu avant les systèmes d'exploitation multitâche préemptifs et qui est revenu à la mode avec des langages plus récents comme Python. Les coroutines sont des tâches qui s'exécutent de façon asynchrone, en pseudo-parallélisme comme avec un noyau moderne. À la différence d'un système préemptif, les coroutines coopèrent pour se partager le processeur : chaque coroutine décide d'« abandonner » le processeur quand elle le souhaite, par exemple lorsqu'elle a terminé une action ou lorsqu'elle demande une entrée/sortie, et non quand ça lui est imposé par le noyau à la suite d'une interruption d'horloge par exemple.

On désire ici réaliser un système de coroutines pour Zorglub33 :

- limité à 10 coroutines au maximum ;
- un identificateur entre 0 et 9 est associé à chaque coroutine ;
- le programme qui initialise le système de coroutines avec la fonction `cr_init` devient de-facto la coroutine 0 dès la fin de la fonction ;
- à l'exception de la coroutine 0, chaque coroutine dispose d'une portion de mémoire de 100 mots pour sa pile d'exécution, démarrant à l'adresse $8000 + i \times 100$ jusqu'à $8000 + (i - 1) \times 100 + 1$ où i est le numéro de la coroutine ;
- la coroutine 0, quant à elle, utilise la pile par défaut ;
- chaque coroutine créée (sauf la coroutine 0) doit démarrer avec les registres A et B initialisés à 0 ;
- lorsqu'une coroutine reprend son exécution après avoir abandonné le processeur (avec `cr_yield` ou `cr_wait`), ses registres sont restaurés comme si elle n'avait jamais abandonné le processeur.

Rédigez en assembleur Zorlub33 les fonctions suivantes :

- `cr_init` : initialise le système de coroutines crée la coroutine 0 pour formaliser le programme actuellement en cours d'exécution (l'appelant de `cr_init`)
- `cr_create` : crée une coroutine pour exécuter le code dont l'adresse est indiquée dans le registre A. Chaque coroutine créée a un numéro de coroutine ;
- `cr_id` : renvoie dans le registre A le numéro de la coroutine actuelle ;
- `cr_yield` : suspend la coroutine actuelle et reprend l'exécution de la prochaine coroutine. Les coroutines sont ordonnancées avec l'algorithme du tourniquet (*round-robin*) ; lorsqu'une coroutine est reprise, elle retrouve ses registres intacts : un appel à cette fonction peut donc être placé au milieu d'un calcul complexe sans que cela affecte le résultat du calcul ;
- `cr_exit` : arrête la coroutine actuelle et reprend l'exécution de la prochaine coroutine ;
- `cr_sleep` : suspend la coroutine actuelle, c'est-à-dire la marque comme ne pouvant continuer son exécution, et reprend l'exécution de la prochaine coroutine ;
- `cr_wakeup` : marque la coroutine dont le numéro est donné dans le registre A comme pouvant reprendre son exécution, sans pour autant changer la coroutine actuelle ;
- `cr_count` : renvoie dans le registre A le nombre de coroutines dans le système (i.e. non terminées), incluant la coroutine 0.

Vous écrirez ces fonctions dans le fichier `coroutine.s` qui contient 3 programmes de test dont les algorithmes et les résultats attendus sont indiqués sous forme de commentaires.

Pour simplifier l'implémentation, on ne demande pas :

- de gérer les erreurs (par exemple : création de plus de 5 coroutines, réveil d'une coroutine qui n'est pas en attente, etc.) ;
- de respecter le protocole d'appel de fonction habituel, notamment de passer les arguments sur la pile et sauvegarde des registres, sauf quand c'est nécessaire pour le fonctionnement des coroutines.