Version 1.01 de cette question. Les ajouts sont en **vert gras**. 

3. Implémentation partielle d'une libraire de thread utilisateur (60 pts)

Les fichiers pour cette question se trouvent dans le répertoire `q3userthread/`.

(**Notez que cette question est très ardue!** Prévoyez plus de temps pour y répondre.)

Comme nous avons vu en classe, il est possible d'implémenter des threads qui sont gérés au niveau utilisateur, plutôt que par le noyau. Le but de cette question est donc de créer une telle libraire en Linux, qui fera de l'ordonnancement coopératif (sans préemption). Votre librairie devra supporter les fonctions suivantes (définies dans `ThreadUtilisateur.h`) :

- initialisation de la libraire via la fonction `int ThreadInit(void);`
- création de threads utilisateurs, via la fonction
  `tid ThreadCreer(void (*fn)(void *), void *arg);` **IMPORTANT! Le nouveau thread créé doit être inséré dans le buffer circulaire à la position qui correspond à la prochaine exécution. Il sera donc intercalé entre le thread en cours, et le thread qui aurait dû être le prochain à être exécuté (comme lorsqu'un thread se réveille, voir plus bas dans la section « Rôle de la fonction `ThreadCeder()` »). À la fin de la fonction `ThreadCreer()`, ne faites pas de `ThreadCeder()`, mais laissez plutôt cette fonction retourner à l'appelant.**
- céder le processeur au prochain thread utilisateur disponible, via la fonction `void ThreadCeder(void)`, avec ordonnancement selon la méthode round-robin (tourniquet);
- terminaison de thread utilisateur via la fonction `void ThreadQuitter(void);`
- attendre qu'un autre thread utilisateur termine, via la fonction
  `int ThreadJoindre(tid ThreadAJoindre);`
- indiquer le numéro de thread en cours d'exécution avec `tid ThreadId(void);`
- faire une pause (et céder l'exécution) avec la fonction `void ThreadDormir(int TempsDormirSeconde);`

Notez comment ces fonctions n'ont jamais comme argument le numéro de thread appelant. Tout comme un système d'exploitation, votre librairie doit savoir en tout temps qui est l'appelant (voir plus bas la variable globale `gpThreadCourant`. Consultez le fichier `ThreadUtilisateur.h` pour une description plus détaillée des fonctions et des codes d'erreurs qu'ils devront retourner.

Pour tous les appels de votre libraire (sauf `ThreadID()`), affichez à l'écran un message comme suit :

```
********** ThreadJoindre(2)  **********
```

Le ficher `ThreadUtilisateur.c` contient déjà le `printf` faisant cet affichage. Donc veuillez ne pas le retirer. Ceci nous aidera à comprendre ce qui se passe.

Présumez qu'il n'y aura pas plus de `MAX_THREADS` threads. Ne faites pas de recyclage de Thread ID. Donc présumez que la fonction `ThreadCreate` ne sera pas appelée plus de (`MAX_THREADS`-2) fois (thread 0 réservé pour *idle*, thread 1 pour le `main`).

Si le programme quitte et qu'il reste des threads bloqués ou qui dorment, ne faites rien de spécial. De toute façon le système d'exploitation est en phase de détruire le tout!

Important! Vous n'avez pas le droit d'utiliser les pthreads ni de libraires existantes similaires! Le but de ce travail est justement de recréer un sous-ensemble des fonctions similaires à ceux trouvés dans ces librairies. C'est un travail qui, pour plusieurs, vous fera sortir de vos sentiers battus. Il vous permettra aussi de comprendre comment un système d'exploitation gère les threads via des structures de données internes.

Avant de commencer à coder, lisez le reste de cette question au complet!

La pierre angulaire de cette libraire sera la structure **`ucontext`**, qui permet de stocker et manipuler un contexte de processeur (donc prendre une image d'une exécution en cours). **Attention!** Vous ne devez jamais copier une structure **`ucontext`**! Il vous faut donc allouer un seul espace pour la stocker (dans une structure similaire à un TCB[1], par exemple), et toujours passer ce contexte à une fonction par son adresse (via un pointeur). La raison est que cette structure **`ucontext`** contient des adresses mémoires qui réfèrent à la structure elle-même. Si vous copiez la structure, ces pointeurs ne seront plus valides, et votre programme risque de planter mystérieusement.

POSIX offre plusieurs fonctions pour utiliser les **`ucontext`**. Les trois fonctions utiles pour ce TP sont :

- `getcontext()`: permet de prendre une copie du contexte actuel. C'est une façon d'initialiser un `ucontext`, en quelque sorte.
- `makecontext()`: pour modifier une structure **`ucontext`** obtenue suite à `getcontext`, comme par exemple modifier le registre EIP pour pointer vers une fonction (qui sera celle exécutée par le thread).
- `swapcontext()` : permet de sauvegarder le contexte en cours et de le remplacer par un autre (pour faire les changements de contexte entre les threads).

Prenez le soin de bien lire les informations sur ces fonctions. Au besoin, regardez les sites suivants pour des exemples d'utilisation de ces fonctions :

- http://pubs.opengroup.org/onlinepubs/009695399/functions/makecontext.html
- http://docs.oracle.com/cd/E19253-01/816-5168/6mbb3hrts/index.html

Les threads devront avoir un état parmi les suivants (définis dans `ThreadUtilisateur.c`):

`THREAD_EXECUTE`	Ce thread est en cours d'utilisation.
`THREAD_PRET`	Ce thread peut être exécuté, mais n'est pas en cours d'utilisation.
`THREAD_BLOQUE`	Ce thread ne peut pas être exécuté. Dans le cas de notre librairie, il attend la fin de l'exécution d'un autre thread sur lequel il a fait un `ThreadJoindre()`, ou bien il est en train de dormir, suivant un appel de `ThreadDormir()`.
`THREAD_TERMINE`	Ce thread a effectué l'appel `ThreadQuitter()`. Il devra être éventuellement détruit selon une approche *garbage collection* qui se déroule lors de `ThreadCeder`( ). En effet, un thread ne peut pas se détruire lui-même.

Structures de données internes à la librairie

Votre solution doit avoir des structures de données internes suivantes (déjà défini pour vous dans `ThreadUtilisateur.c`)

- Un **buffer circulaire** contenant tous les threads qui sont dans les états `THREAD_EXECUTE`, `THREAD_PRET` ou `THREAD_TERMINE`. Ce buffer circulaire sera utilisé pour faire l'ordonnancement tourniquet. Le pointeur `gpNextToExecuteInCircularBuffer` pointe vers le prochain thread à exécuter.
- Une **file d'attente** `gpWaitTimerList` qui est une liste chaînée simple de tous les processus qui dorment suite à un appel de `ThreadDormir()`. L'état de ces threads sera `THREAD_BLOQUE`.
- Chaque TCB aura aussi **une liste chaînée simple** (`pWaitListJoinedThreads`) qui donne la liste des threads qui ont fait un `ThreadJoindre` sur ce thread. L'état de ces threads qui ont fait un join sera aussi `THREAD_BLOQUE`.

[1] TCB = Thread Control Block. Voir `ThreadUtilisateur.c` pour un exemple possible.

Selon les différentes actions prises par les threads, elles seront déplacées dans l'une des trois structures nommées précédemment. Je vous impose ces types de structures car elles se rapprochent de ce que l'on trouve dans un système d'exploitation. Vous pouvez les implémenter comme vous le voulez, en vous assurant que le tout compile dans la machine virtuelle du cours.

Quelques conseils

Initialisation de la libraire par `ThreadInit()`

Il faut y créer tout d'abord un TCB (Thread Control Block, initialisé correctement, voir plus bas pour plus de description) à l'aide d'un **`malloc`**[2] pour exécuter la fonction `IdleThreadFunction`. Ce thread ne fait à peu près rien sauf dormir pendant 250 ms. Ce thread nous assure qu'il y a au moins toujours un thread en activité, pour permettre au système de se tourner les pouces. Ceci nous assure que la fonction `ThreadCeder` est appelée périodiquement, car cette dernière va vérifier s'il y a un thread qui doit être réveillé (période de sommeil expirée). Il faudra aussi créer un TCB pour le thread du main, car on doit stocker le contexte du main comme si c'était un thread utilisateur lors des changements de contexte. À la différence des autres threads, ce TCB pour le main n'aura pas besoin d'une nouvelle pile. Le thread `IdleThreadFunction` devra avoir comme ID la valeur 0, le thread main 1, et les threads créés subséquemment les valeurs 2, 3, etc.

Création d'un thread, pas-à-pas, pour la fonction `ThreadCreer()`

Afin de vous guider un peu, voici les opérations (un peu ésotériques!) à faire dans votre libraire pour créer un thread utilisateur de toutes pièces :

1. Créer, avec **`malloc`**, une structure de donnée **`TCB`** qui contiendra un **`ucontext`**. Les autres champs de cette structure **`TCB`** sont à votre choix[3], pour vous permettre de gérer ces threads. C'est en quelque sorte votre version du Thread Control Block qui est dans un noyau.
2. Initialiser le **`ucontext`** dans cette structure (disons **`ctx`**) par le contexte actuel via la ligne de code suivante : **`getcontext(&TCB.ctx);`**
   À ce moment-là, **`TCB.ctx`** contiendra une copie du contexte en cours. Ce sera la base à partir de laquelle nous allons créer un nouveau thread.
3. Chaque thread doit posséder sa propre pile (sauf le thread 1 pour le main, puisqu'il a déjà sa pile). Vous devez donc allouer un bloc mémoire pour ce nouveau thread
   `char *pile = malloc(TAILLE_PILE);`
4. Il faut maintenant affecter cette nouvelle pile au thread que vous êtes en train de créer. Cela se fera en écrasant la valeur du pointeur de pile dans **`TCB.ctx`** :
   `TCB.ctx.uc_stack.ss_sp = pile;`
   Cette opération chargera le registre de pointeur de pile du processeur (ESP) vers cette nouvelle pile, lorsque l'on mettra ce contexte sur le processeur. Notez que cette opération n'affecte pas le contexte en cours.
5. Il faut aussi spécifier la taille de cette pile :
   `TCB.ctx.uc_stack.ss_size = TAILLE_PILE;`
6. Il nous reste maintenant à affecter au contexte la fonction à exécuter lorsqu'il sera activé sur le processeur. La fonction **`makecontext`** arrive à la rescousse :
   `makecontext(&TCB.ctx, (void *)pFuncThread`[4]**`, 1, pArg);`**

[2] `malloc` alloue de la mémoire sur le tas, donc en dehors de la pile.
[3] ou vous pouvez réutiliser ceux que j'ai déjà défini dans le fichier `ThreadUtilisateur.c` fourni.
[4] Notez qu'il n'y a pas de symbole `&` devant `pFuncThread`, car c'est déjà un pointeur. Si vous ajoutez un `&` devant

où **pFuncThread** est un pointeur de la fonction à exécuter, et **pArg** un pointeur sur **void** qui contient les arguments à passer à cette fonction.

7. N'oubliez-pas d'initialiser l'état de ce thread à **THREAD_PRET**, et toutes autres variables dans le **TCB** associées à votre thread utilisateur, comme son numéro de thread.
8. Ce thread peut maintenant être inséré dans le buffer circulaire et est prêt à être exécuté!

Rôle de la fonction `ThreadCeder()`

Du point de vue utilisateur, la fonction `ThreadCeder()` ne sert qu'à permettre à un autre thread de s'exécuter. De notre point de vue de concepteur de cette librairie, c'est cette fonction qui, à la manière du top d'horloge d'un système d'exploitation, sera au cœur de l'action! En effet, cette fonction devra accomplir les tâches internes suivantes (et complètement invisibles à l'utilisateur) :

- Consulter la liste `gpWaitTimerList` pour voir si un ou des threads doivent être réveillés. La granularité est de l'ordre de la seconde (j'ai utilisé `time()`). Si un thread doit être réveillé, il faudra 1) le retirer de cette liste 2) changer son état à `THREAD_PRET` et 3) le placer dans le buffer circulaire de sorte qu'il soit le prochain à être exécuté. S'il y en a plus d'un, l'ordre importe peu entre eux, pour autant qu'ils soient tous être ordonnancés en premier (i.e. avant le thread qui aurait dû être ordonnancé s'il n'y avait pas eu ces threads de réveillé).
- Faire du *garbage collection*[5]. Si le prochain thread est marqué comme `THREAD_TERMINE`, il vous faudra le retirer du buffer circulaire et le détruire (désallouer sa pile, son TCB et autres structures que vous emploierez). Vous continuez ainsi jusqu'au prochain processus prêt à être exécuté.
- Sélectionner le prochain thread à exécuter, et faire le changement de contexte pour passer d'un thread à un autre. La fonction **`swapcontext()`** échangera un contexte sauvegardé (celui que vous voulez faire tourner) pour celui en cours. La grande difficulté ici est de s'assurer que vous sauvegardez le contexte en cours dans le bon TCB (sinon vous allez avoir des comportements assez étranges!). Assurez-vous de changer l'état du prochain thread qui sera exécuté à l'état `THREAD_EXECUTE` et que l'ancien thread est à l'état `THREAD_PRET`. En aucun cas vous ne devez exécuter un thread qui est dans l'état `THREAD_TERMINE`. N'oubliez-pas d'ajuster la variable globale `gpThreadCourant` pour la faire pointer vers le TCB du prochain thread juste avant le changement de contexte avec `swapcontext`. En tout temps, le système doit savoir quel thread il est/deviendra…

Aussi, à chaque appel de `ThreadCeder()`,affichez à l'écran le contenu du buffer circulaire et de la file d'attente `gpWaitTimerList`, selon le format suivant :

```
-----  Etat de l'ordonnanceur avec 3 threads -----
| prochain->ThreadID:3  État:P    WaitList
|           ThreadID:2  État:P    WaitList-->(1)
|           ThreadID:0  État:P *Special Idle Thread*   WaitList
-----   Liste des threads qui dorment, epoch time=1455081102  -----
|           ThreadID:4 État:B WakeTime=1455081105  WaitList
-------------------------------------------------
```

où la lettre correspondant à l'état est E: `THREAD_EXECUTE`, P: `THREAD_PRET`, B: `THREAD_BLOQUE`, T: `THREAD_TERMINE`. De plus, vous devez afficher la liste des threads en attente sur un autre thread

---

`pFuncThread`, vous aurez un *segmentation fault* lors de l'opération `swapcontext`, ce qui vous causera du grattage de tête.

[5] Je me répète ici, mais il est impossible pour un thread de s'autodétruire. En effet, comment pourrait-il désallouer sa propre pile? Ce serait comme tenter de retirer un tapis sous nos propres pieds, avec des conséquences fâcheuses comme un culbutage.

avec l'affichage `WaitList-->`. Par exemple, on voit dans l'affichage ci-dessus que le thread 1 attend après la fin du thread 2 (car le thread 1 a précédemment fait `ThreadJoindre(2)`. Pour savoir quel processus sera le prochain à être exécuté dans le buffer circulaire (marqué `prochain->`), j'utilise un pointeur que j'ai nommé `gpNextToExecuteInCircularBuffer`. Vous devez aussi indiquer par `*Special Idle Thread*` que le thread 0 est le *Idle thread*. Simplement pour m'assurer que vous compreniez mieux ce concept de *Thread 0*/*Idle thread*, présent dans plusieurs systèmes d'exploitation.

Important! N'oubliez-pas que toutes les variables locales (les variables dans la fonction `ThreadCeder`) sont sur une pile. Donc, l'utilisation de variables locales avec un `swapcontext` doit être faite délicatement, sinon votre programme aura des comportements aux apparences étranges. Il est donc important de bien mettre à jour toutes les données globales (structures et pointeurs) avant de faire ce `swapcontext`, car à ce moment-là l'exécution sera téléportée dans une autre pile et contexte (aussi appelé un *longjump/longjmp*), qui sera fort probablement la suite de `ThreadCeder` après le `swapcontext` d'un autre thread.

Aussi, cette fonction `ThreadCeder()` sera utilisée par d'autres fonctions de votre librairie. Par exemple, dans la fonction `ThreadDormir()`, après avoir retiré le thread appelant du buffer circulaire et l'avoir placé dans la file d'attente, vous devez faire un `ThreadCeder()`. Votre ordonnanceur choisira alors un prochain thread et fera le changement de contexte. *The show must go on!*

Fin d'un thread

IMPORTANT! Assumez que tous les threads font l'appel **`ThreadQuitte()`** pour indiquer qu'ils ont quitté. Il est un peu compliqué (sans être impossible) de s'assurer qu'après l'exécution d'un thread, votre librairie fasse le nettoyage approprié si on oublie de faire `ThreadQuitte()`. Pour les curieux, demandez-moi la solution.

À fournir dans le rapport

- Le listing du code;
- Indiquez si certaines fonctions n'ont pas été implémentés ou sont susceptibles de planter;
- La sortie d'écran suite à l'exécution du code `TestThread.c`.

N'oubliez pas d'inclure aussi le code source dans le fichier .zip soumis afin que nous puissions recompiler votre solution.

Sortie d'écran

Pour vous aider, voici une partie de la sortie d'écran de mon programme lors de l'exécution du programme `TestThread` (`TestThread.cpp`) linké avec ma librairie :

```
  ********************************** ThreadInit()  **********************************

  ********************************** ThreadCreer(0x8048c40,(nil))  ********************************
(0.602) Main: Le thread ID du main est 1.

  ********************************** ThreadCreer(0x804891d,0xbf8fb3e8)  ********************************
(0.602) Main: Le thread avec ID 2 a été créé.

  ********************************** ThreadCreer(0x804891d,0xbf8fb3f4)  ********************************
(0.602) Main: Le thread avec ID 3 a été créé.

  ********************************** ThreadCreer(0x804891d,0xbf8fb400)  ********************************
(0.602) Main: Le thread avec ID 4 a été créé.
(0.602) Main: Je joins le thread ID 2

  ********************************** ThreadJoindre(2)  **********************************

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 4 threads -----
   | prochain->ThreadID:4  État:P    WaitList
   |           ThreadID:3  État:P    WaitList
   |           ThreadID:2  État:P    WaitList-->(1)
   |           ThreadID:0  État:P *Special Idle Thread*   WaitList
   -----   Liste des threads qui dorment, epoch time=1455134060  -----
   ------------------------------------------------

(0.602) Thread4: Je tourne avec une variable sur la pile à 0x0x8c8875c.

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 4 threads -----
   | prochain->ThreadID:3  État:P    WaitList
   |           ThreadID:2  État:P    WaitList-->(1)
   |           ThreadID:0  État:P *Special Idle Thread*   WaitList
   |           ThreadID:4  État:E    WaitList
   -----   Liste des threads qui dorment, epoch time=1455134060  -----
   ------------------------------------------------

(0.643) Thread3: Je tourne avec une variable sur la pile à 0x0x8c865dc.

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 4 threads -----
   | prochain->ThreadID:2  État:P    WaitList-->(1)
   |           ThreadID:0  État:P *Special Idle Thread*   WaitList
   |           ThreadID:4  État:P    WaitList
   |           ThreadID:3  État:E    WaitList
   -----   Liste des threads qui dorment, epoch time=1455134060  -----
   ------------------------------------------------

(0.668) Thread2: Je tourne avec une variable sur la pile à 0x0x8c8445c.

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 4 threads -----
   | prochain->ThreadID:0  État:P *Special Idle Thread*   WaitList
   |           ThreadID:4  État:P    WaitList
   |           ThreadID:3  État:P    WaitList
   |           ThreadID:2  État:E    WaitList-->(1)
   -----   Liste des threads qui dorment, epoch time=1455134060  -----
```

```
   ------------------------------------------------

                #########  Idle Thread 0 s'exécute et va prendre une pose de 250 ms... #######

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 4 threads -----
   | prochain->ThreadID:4  État:P    WaitList
   |           ThreadID:3  État:P    WaitList
   |           ThreadID:2  État:P    WaitList-->(1)
   |           ThreadID:0  État:E *Special Idle Thread*   WaitList
   -----   Liste des threads qui dorment, epoch time=1455134060  -----
   ------------------------------------------------
```

…

Je saute des pas d'exécution

…

```
                #########  Idle Thread 0 s'exécute et va prendre une pose de 250 ms... #######

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 2 threads -----
   | prochain->ThreadID:3  État:P    WaitList-->(1)
   |           ThreadID:0  État:E *Special Idle Thread*   WaitList
   -----   Liste des threads qui dorment, epoch time=1455134069  -----
   |           ThreadID:4 État:B WakeTime=1455134070  WaitList
   ------------------------------------------------

(9.213) Thread3: Je QUITTE!
  ********************************** ThreadQuitter(3)  **********************************
 ThreadQuitter: je reveille le thread 1

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 3 threads -----
   | prochain->ThreadID:1  État:P    WaitList
   |           ThreadID:0  État:P *Special Idle Thread*   WaitList
   |           ThreadID:3  État:T    WaitList
   -----   Liste des threads qui dorment, epoch time=1455134069  -----
   |           ThreadID:4 État:B WakeTime=1455134070  WaitList
   ------------------------------------------------

(9.213) Main: Le thread ID 3 a terminé!
(9.213) Main: Je joins le thread ID 4

  ********************************** ThreadJoindre(4)  **********************************

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 2 threads -----
   | prochain->ThreadID:0  État:P *Special Idle Thread*   WaitList
   |           ThreadID:3  État:T    WaitList
   -----   Liste des threads qui dorment, epoch time=1455134069  -----
   |           ThreadID:4 État:B WakeTime=1455134070  WaitList-->(1)
   ------------------------------------------------

                #########  Idle Thread 0 s'exécute et va prendre une pose de 250 ms... #######

  ********************************** ThreadCeder()  **********************************
   -----  Etat de l'ordonnanceur avec 2 threads -----
   | prochain->ThreadID:3  État:T    WaitList
   |           ThreadID:0  État:E *Special Idle Thread*   WaitList
   -----   Liste des threads qui dorment, epoch time=1455134069  -----
   |           ThreadID:4 État:B WakeTime=1455134070  WaitList-->(1)
   ------------------------------------------------
```

```
ThreadCeder: Garbage collection sur le thread 3
              #########  Idle Thread 0 s'exécute et va prendre une pose de 250 ms... #######

 ********************************** ThreadCeder()  **********************************
  -----  Etat de l'ordonnanceur avec 1 threads -----
  | prochain->ThreadID:0  État:E *Special Idle Thread*   WaitList
  -----   Liste des threads qui dorment, epoch time=1455134069  -----
  |           ThreadID:4 État:B WakeTime=1455134070  WaitList-->(1)
  ------------------------------------------------
```

…
et ça continue…

Mon implémentation, à titre indicatif seulement!

Dans le fichier **`ThreadUtilisateurs.c`** j'ai laissé quelques vestiges de mon implémentation, afin de vous inspirer. Si vous le désirez, vous pouvez changer les structures de données (TCB) et les variables globales. Au total ma solution dans le fichier **`ThreadUtilisateurs.c`** faisait environ 450 lignes.

J'ai utilisé des variables globales pour stocker les informations suivantes :

`static TCB *gpThreadCourant;` Le pointeur sur le TCB du thread en cours d'utilisation

`static TCB *gpNextToExecuteInCircularBuffer;` Le pointeur sur le prochain TCB à exécuter.

`static int gNumberOfThreadInCircularBuffer=0;` Le nombre de TCB dans la file d'ordonnancement

`static int gNextThreadIDToAllocate=0;` Le numéro d'identification du prochain thread qui sera créé.

`static WaitList *gpWaitTimerList = NULL;` La tête de la file d'attente pour les threads qui dorment.

`static TCB *gThreadTable[MAX_THREADS];` Un tableau pour stocker le mappage entre un numéro de thread (l'index du tableau) et son pointeur (le contenu du tableau), utilisé par la fonction `ThreadID()`.

Comme il n'y a pas de préemption, il n'est pas nécessaire d'utiliser un mutex pour protéger ces variables. (à moins que vous ne faisiez un `ThreadCeder()` en pleine section critique!)

Pour conserver la trace des threads qui ont fait un `ThreadJoindre()`, j'ai utilisé une liste chaînée dans le TCB (`pWaitListJoinedThreads`) qui indiquait tous les threads bloqués qui sont en attente. Lorsque le thread invoquait `ThreadQuitter()`, il parcourait cette liste pour réactiver tous les threads en attente sur lui-même.

Aussi, commencez par les fonctions les plus simples, comme `ThreadInit()`, `ThreadCreate()` `ThreadID()` et `ThreadCeder()`.Quand vous aurez bien maîtrisé l'art du changement de contexte, ajoutez `ThreadQuitter()` (et le *garbage collection*), `ThreadJoindre()` et finalement `ThreadDormir()`.