Manuel de référence

Généré par Doxygen 1.2.17

Mon Jan 5 16 :22 :05 2004

Table des matières

1 FRED : un noyau minimal de threads

Auteur :
J. Briat

Date :
Janvier 2004

FRED est un noyau de thread minimal à but pédagogique. Il donne un exemple d'implantation dit N : :1 ou en mode utilisateur. Un processus lourd (process UNIX ici) est vu comme une machine monoprocesseur dont on va partager les ressources (temps d'exécution et mémoire) entre plusieurs threads. Un thread sera simplement l'exécution d'une procédure C. L'exécution des threads par le processeur virtuel (processus lourd) se fera périodiquement à tour de rôle (temps partagé) ou au gré des synchronisations entre ces threads.

L'étudiant trouvera une description de FRED à travers les sections qui suivent : La description de l'interface applicative(**L'interface applicative de FRED** (p. 1)) :comment écrire un programme concurrent FRED, comment lancer des threads etc.. Il trouvera aussi une description de l'implémentation(**Implantation de FRED** (p. 6)) et des mécanismes de base nécessaires. Dans la version html de ce document, il pourra naviguer à son gré dans la description des différents modules de FRED.

1.1 L'interface applicative de FRED

FRED offre les fonctionnalités primitives d'un noyau de threads gérées en temps partagé. Un noyau de thread réaliste doit offrir des fonctions plus élaborées. Toutes ces fonctions ne sont cependant que des extensions des mécanismes de base de FRED. Ces fonctions sont présentées en plusieurs sections :

- **Initialisation & terminaison** (p. 2) : comment définir un programme concurrent
- **Gestion des threads** (p. 3) : comment créer des threads
- **Exclusion mutuelle** (p. 3) : comment faire une section critique
- **File d'attente** (p. 4) : comment synchroniser des threads
- **Gestion mémoire** (p. 4) : comment allouer/libérer de la mémoire libre
- **Entrée/Sortie** (p. 4) : comment lire/écrire élémentaire dans un fichier ou flot d'E/S
- **Exemples d'utilisation** (p. 5) : des exemples de réalisation d'objet de synchronisation (**Un exemple d'outil de synchronisation : le sémaphore** (p. 5)) ou d'algorithme de synchronisation *classique* (**Les N philosophes** (p. 6))

1.1.1 Initialisation & terminaison

Pour utiliser FRED, il faut inclure le fichier **fred.h** pour disposer de toutes les déclaration de types et profils de procédures de l'interface. Il est aussi nécessaire dans la procédure initiale **main**() du programme principal concurrent d'initialiser les variables statiques de FRED. C'est le rôle de **FredInitialize**() (p. 14). Puis le programmeur peut initialiser ses propres variables globales et lancer ses threads initiaux, se synchroniser etc.... **FredWaitTerminate**() (p. 14) lui permet d'attendre la terminaison des threads initiaux (et des threads créés par ceux-ci et etc....). Quand la procédure **main**() termine, tout le programme concurrent termine.

```
#include <fred.h>
int main( int argc, char** argv)
{/*
    FRED n'est pas utilisable.
    Seul du code C séquentiel est exécutable
 */
  <....>
FredInitialize();
 /*
    FRED est partiellement utilisable.
    On peut créer des threads, intialiser des objets satiques,
    se synchroniser ....
 */
<....>
FredWaitTerminate();
 /*
    Attente de la terminaison de tous les threads.
    A la reprise, FRED n'est plus utilisable.
    Seul du code C séquentiel est exécutable.
 */
 <....>
}
```

N'importe quel thread peut terminer brutalement le programme concurrent par **FredAbort**() (p. 14) en passant en paramètre une chaine de caractère précisant la raison de l'arrêt brutal.

Voir également :

Interface d'initialisation & terminaison

1.1.2 Gestion des threads

Un thread est l'exécution concurrente d'une procedure C à **un** paramètre de type (void*) et **un** résultat de type (void*). Son type est défini par **FredRunnable** (p. 15). On rappelle que tout pointeur (T*)étant **"castable"** en (void*) et réciproquement, le type (void *) veut dire référence sur n'importe quel type. Pour s'exécuter, un thread a besoin d'une pile pour ses contextes (trames) de procédure. La création d'un thread nécessite donc de donner de :

- une procédure de type **FredRunnable** (p. 15)
- une taille de pile
- un paramètre de type (void*)
- un endoit où ranger un résultat de type (void*) soit un paramètre de type ((void*)*). Si aucun résultat n'est souhaité, mettre (void**)0.

La procédure **FredThreadSpawn**() (p. 17) crée et lance donc un thread (c.à.d. le met dans la file des threads prêts à s'exécuter).

Un thread se termine par terminaison normale de sa procédure initiale ou par suicide. La procédure **FredThreadExit**() (p. 16) permet à un thread de se terminer en rendant un résultat de type (void*).

Un thread peut connaitre son identité (reférence sur son descripteur (type **FredThread** (p. 15))) par la fonction **FredThreadCurrent**() (p. 16).

Le thread courant peut spontanément abandonner le processeur au profit d'un autre thread prêt au sens des règles d'ordonnancement par **FredThreadYield**() (p. 18).

Voir également :
Interface de gestion de threads

1.1.3 Exclusion mutuelle

FRED n'offre qu'un mécanisme d'exclusion mutuelle basé sur les mécanismes matériel : inhibition des interruption sur monoprocesseur qui doit être étendu par un verrouillage de type "TestAndSet" sur multiprocesseur. L'utilisation d'un verrouillage matériel n'est utilisable que sur des durées très courtes pour éviter de retarder ou perdre des signaux externes ou bloquer les autres processeurs(cas multiprocesseur). Dès que la durée d'exclusion mutuelle devient importante, il faut recourir à des mécanismes de synchronisation plus élaborés (sémaphore, verrou Posix etc..).

L'exclusion mutuelle repose sur les 2 procédures **FredMutexOn**() (p. 19) et **FredMutexOff**() (p. 19).

Avertissement :
Il est interdit d'imbriquer les sections critiques.

```
/* code incorrect */
....FredMutexOn();....; FredMutexOn();....FredMutexOff();....FredMutexOff();....
```

Voir également :
Interface d'exclusion mutuelle

1.1.4 File d'attente

L'exclusion mutuelle ne suffit pas pour exprimer toutes les relations de dépendance et de synchronisation entre threads. Il est nécessaire de bloquer un thread si sa continuation falsifierait un **invariant d'état global** et, réciproquement, de réveiller les threads dès que leurs continuations préservent cet invariant. FRED offre le type **FredQueue** (p. 16) qui permet de définir une file d'attente de threads gérée selon la statégie FIFO. On initialise une queue par **FredQueueInit**() (p. 20).

Le thread courant peut se bloquer dans une file d'attente c.à.d. il s'inscrit en dernier dans cette file et donne le processeur à un autre thread prêt. C'est la procédure **FredThreadSuspend**() (p. 21).

Une fois bloqué dans une file, un thread peut être sorti de la file par **FredQueue-GetFirst**() (p. 20), remis dans une autre file par **FredQueuePutLast**() (p. 20). On peut réveiller le premier d'une file d'attente c.à.d. le remettre dans la file des threads prêts par **FredThreadWakeUp**() (p. 21).

Utilisées **nécessairement** dans une portion de code en exclusion mutuelle (entre **FredMaskOn**() (p. 40) et **FredMaskOff**() (p. 39)), ces procédures permettent de réaliser des opérateurs de sunchronisation plus élaborés (**Un exemple d'outil de synchronisation : le sémaphore** (p. 5)).

Voir également :
  Interface de gestion de file d'attente

1.1.5 Gestion mémoire

La création des threads comme la programmation concurrente en C nécessitent d'allouer et libérer dynamiquement des blocs de mémoire. Les procédures d'allocation et libération modifient la description de l'état d'occupation de la mémoire. Ce sont des procédures qui doivent s'exécuter en exclusion mutuelle. FRED fournit un carrossage de la gestion mémoire standard de C pour assurer l'atomicité de l'allocation/ libération des blocs mémoires.

FredAtomicAlloc() (p. 23),**FredAtomicCalloc**() (p. 23) et **FredAtomic-Free**() (p. 24)sont fonctionnellement identiques aux procédures malloc(),calloc() et free() de la libC.

Les trois macros **FredNEW**() (p. 22), **FredNEW_VECTOR**() (p. 23) et **FredDELETE**() (p. 22) sont de simples carrossages des précédentes.

Voir également :
  Interface de gestion mémoire

1.1.6 Entrée/Sortie

Pour des raisons identiques à la gestion mémoire, l'accès aux fichiers par les threads doit aussi être synchronisé. FRED ne fournit des procédures atomiques que pour les procédures de type printf() et scanf() de la libC.

Voir également :
Interface d'entrée/Sortie

1.2 Exemples d'utilisation

L'exclusion mutuelle par masquage et la gestion de files de threads sont suffisantes pour reconstruire des opérateurs de synchronisation plus souples et evitant de masquer les interruption trop longtemps. C'est le cas du sémaphore de Dijkstra.

On peut ensuite traiter un problème classique de synchronisation comme les N philosophes.

1.2.1 Un exemple d'outil de synchronisation : le sémaphore

Un sémaphore est un simple sac où l'on prend ou remet des jetons un à un. Un thread demandant un jeton doit attendre qu'il y en ait un de disponible. Tout thread déposant un jeton doit réveiller le plus ancien thread en attente de jeton s'il en existe. Le sémaphore doit être déclaré de type **FredSemaphore** (p. 50), alloué en variable globale ou sur le tas. Dans tousles cas, il doit être initialisé à une valeur >=0 via **FredSemaphoreInit**() (p. 28).

```
FredSemaphore GlobalS ; /* sémaphore global */
struct S1 {             /* struct globale */
  Semaphore member; /* avec sémaphore membre */
  ...
}

void exemple()
{
  S1 * s;
  FredSemaphore LocalS ; /* sémaphore local */
  FredSemaphore*  DynaS ; /* sémaphore dynamique */
  .....
  FredSemaphoreInit(&LocalS,0); /* initialisation */
  ....
  DynaS = FredNEW(FredSemaphore); /* allocation */
  FredSemaphoreInit(DynaS,3); /* initialisation */
  .....
  FredSemaphoreInit(&GlobalS,25); /* initialisation */
  .....
  s= FredNEW(S1); /* allocation d'une structure */
  FredSemaphoreInit(&(s->member),33); /* initialisation du membre*/
  .....
}
```

Il peut ensuite être utilisé pour prendre **FredSemaphoreP**() (p. 29) ou déposer un jeton **FredSemaphoreV**() (p. 29)

L'implantation nécessite un compteur et une queue de threads en attente (cf. **FredSemaphore** (p. 50)). Le compteur représente le nombre de jetons si le compteur est >=0 et le nombre de threads en attente s'il est <0. Le test du compteur, sa modification et la manipulation de la file d'attente doivent être atomiques et se font donc masqués. (cf. **Interface sémaphore** (p. 27))

1.2.2 Les N philosophes

Le programme principal initie :
– les sémaphores représentant les fourchettes
– les philosophes

```
#define NBphilo
FredThread philosophe[NBphilo];
FredSemaphore fourchette[NBphilo];
int main ( int argc, char** argv)
{
  FredInitialize();
  for( i=0;i<NBphilo;i++)
    FredSemaphoreInit(&fourchette[i],1);
  for( i=0;i<NBphilo;i++)
   philosophe[i]= FredThreadSpawn(20000,codephilo,(void*)i,(void**)0);
  FredWaitTerminate();
  /*  C'est termine */
  return 0;
}/
```

L'interblocage se produit dès que les philosophes utilisent un ordre horaire ou antihoraire pour prendre leurs fourchettes. Par exemple un philosophe **i** prend la fourchette **i** puis la fourchette **(i+1)%NBphilo**. La stratégie d'évitement du blocage est l'allocation des ressources selon un ordre total universel. Ceci veut dire que si les philosophes doivent prendre les fourchettes par numéro toujours croissant ou décroissant. Ceci implique que le philosophe de rang le plus grand **(i==NBphilo-1)** prendra la fourchette **0==(i+1)%NBphilo** puis la fourchette **i** c'est à dire un comportement inverse des autres.

```
void* codephilo(void * x)
{ int j;
  int i = (int)x; /* numero du philosophe */
  for( j=0;j<100;j++){ /* 100 iterations de penser/manger */
     /* ça pense ..*/
     /* ça pense à manger ..*/
    if(i<(NBphilo-1)){ /* un droitier */
      FredSemaphoreP(&fourchette[i]);FredSemaphoreP(&fourchette[i+1]);
       /* ça mange ..*/
      FredSemaphoreV(&fourchette[i]);FredSemaphoreV(&fourchette[i+1]);
    }else{ /* le gaucher */
      FredSemaphoreP(&fourchette[0]);FredSemaphoreP(&fourchette[NBphilo-1]);
      /* ça mange ..*/
      FredSemaphoreV(&fourchette[0]);FredSemaphoreV(&fourchette[NBphilo-1]);
    }
  }
  return (void*) 0; /* suicide */
}
```

Pour plus de détails, voir le fichier exemple des **N philosophes.** (p. 58)

1.3 Implantation de FRED

Le principe d'implantation d'un noyau de gestion de thread sur un monoprocesseur est d'entrelacer l'exécution des procédures associées à des threads au gré de synchronisation explicite (sémaphore, verrou,..) ou des interruptions (temps partagé, E/S). Il faut donc être capable de trier entre les threads **suspendus** en attente d'un signal externe (E/S, fin de quantum,..) ou interne (sémaphore passant, réveil par un autre thread,..), **ceux** prêts à tourner et celui qui s'exécute sur le processeur : le thread **actif** ou **courant**. La technique classique est l'utilisation de listes que l'on appelle **queues** ou **files** d'attente (**Gestion de listes** (p. 7)).

L'autre mécanisme nécessaire est le mécanisme qui permet de passer d'une exécution de procédure à une autre puis faire l'opération inverse. Pour ce faire, il suffit de sauvegarder l'état du processeur (registres) d'un thread et de relancer un autre thread depuis un état sauvé. Il faut de plus savoir créer un nouvel état d'exécution lors de la création d'un nouveau thread. A tout thread sera associée une zone de sauvegarde pour ses registres (compteur ordinal, stack pointeur, registres généraux etc...). La taille et l'organisation de cette zone dépend donc du processeur et de la façon de sauvegarder les registres. Pour passer d'un thread à un autre, il suffira de sauver les registres dans la zone du thread en cours d'exécution puis de recharger les registres à partir de la zone de sauvegarde d'un autre thread qui reprend alors sont exécution(**Commutation de contexte** (p. 8)).

Enfin,il faut disposer d'un mécanisme permettant de faire ces commutation entre threads de façon atomique. Comme sur un monoprocesseur, la seule possibilité pour empêcher qu'une suite d'instructions de s'exécuter complètement est une interruption, l'atomicité est simplement réalisée en inhibant les interruptions au début de cette séquence(**Inhibition des interruptions** (p. 9))

FRED est un noyau où les threads sont gérés en temps partagé. Il faut donc d'une part récupérer des interruptions(**Traitement des interruptions** (p. 8)) et gérer un mécanisme de décomptage du temps pour provoquer une interruption périodique permettant de changer le thread actif(**Gestion du décompteur** (p. 9))

La réalisation et mise en oeuvre du noyau de threads utilise des fonctions d'allocation mémoire et d'E/S. Les fonctions offertes par la bibliothèque standard C n'ont pas toujours été prévues pour être partagées entre des threads. Il est prudent de les rendre compatibles (**thread safe**)avec les threads (**E/S et gestion de mémoire "thread-safe"** (p. 10)).

L'implantation d'un noyau de thread est alors l'utilisation coordonnée de toutes ces fonctions(**Gestion des threads** (p. 10)).

1.3.1 Gestion de listes

La gestion de liste proposée est une liste **circulaire doublement chainée**. Ceci veut dire que toute cellule (de type **FredCell** (p. 49)) de liste contient un double chainage c'est à dire un pointeur sur la cellule d'avant et celle d'après. La première cellule et la dernière cellule sont reliées. La liste (de type **FredList** (p. 49)) pointe sur un élément qui est considéré comme le dernier élément de la liste. L'élément qui le suit est donc le premier élément de la liste. L'élément qui le précède est donc l'avant-dernier élément de la liste. Pour plus de détails, voir **Interface de gestion de listes** (p. 30)

1.3.2 Commutation de contexte

Du point de vue de l'entrelacement des exécutions sur le processeur des différentes procédures que sont les threads, un descripteur de thread doit inclure une zone de sauvegardes pour ses registres (compteur ordinal, stack pointeur, registres généraux etc...). La taille et l'organisation de cette zone dépend donc du processeur et de la façon de sauvegarder les registres. Pour passer d'une exécution de thread à un autre, il suffira de sauver les registres dans la zone du thread en cours d'exécution puis de recharger les registres à partir de la zone de sauvegarde d'un autre thread qui reprend alors son exécution.

En plus des opération de sauvegarde restauration, il faut pouvoir initialiser une zone de sauvegarde de façon qu'un thread puisse démarrer correctement sa procédure initiale avec sa pile.

La gestion de Contexte est proposée en plusieurs variantes pour des systèmes UNIX . La première est basée sur les procédures `setjmp()-\c` lonjmp() qui sont fournies avec la librairie standard C et qui réalisent une forme de sauvegarde/restauration de contexte permettant d'écrire une commutation contexte de façςn portable. Par contre, l'initialisation d'un contexte reste à faire en fonction de la machine hôte et de l'implantation de la structure "jmpbuf" utilisée par la lib C. On trouvera une variation à destination de Linux IA32 (**Gestion de contexte pour Linux IA32 par setjmp/longjmp** (p. 33)) et l'autre d'un Sparc32 sous Solaris (**Gestion de contexte pour Solaris Sparc32 par setjmp/longjmp** (p. 37))

La seconde façon consiste à écrire directement la commutation de contexte en code natif pour une machine donnée.L'exemple (**Gestion de contexte pour Linux IA32 (code natif)** (p. 35)) proposé est à destination de Linux et l'architecture IA32 d'Intel (386/486/Pentium etc.et clones..). La méthode proposée sauve tous les registres dans la pile d'un thread et sauve donc le seul pointeur de pile dans le contexte. Ces procédures dépendent de la machine hôte mais aussi des conventions de passage de paramètres entre procédures C, des prologues et épilogues des procédures générés par le compilateur C. Pour les connaitre, il faut analyser le code assembleur engendré par le compilateur (option -S). La version proposé est conforme au code généré par le compilateur gcc 3.2

Le choix de l'implantation à utiliser pour FRED est fixé par une macro variable du préprocesseur C qui permet d'inclure l'implantation choisi. (cf le fichier makefile et le début du fichier **contextswitch.h**)

1.3.3 Traitement des interruptions

La gestion des autorisations/inhibitions des interruptions est nécessaire pour plusieurs raisons :
- l'utilisation des interruptions d'une horloge permet d'implanter une allocation du processeur en temps partagé et une gestion d'échéancier etc..
- l'utilisation des interruptions permet de réagir aux évènements extérieurs (E/S)
- le masquage des interruptions est le seul moyen INITIAL d'assurer l'atomicité d'exécution d'une séquence de code

Les interruptions issues de l'environnement sont transmises par le système aux processus lourds utilisateurs selon un modèle voisin du mécanisme matériel. L'occurrence d'un signal répéré par son numéro provoque l'exécution d'une routine de traitement qui lui a été associée(attachée). La sauvegarde des registres et le masquage contre le signal est fait préalablement à l'appel de la routine de traitement. La restauration de l'état antérieure est fait au retour de cette routine de traitement.

Une procédure de traitement de signal est une procédure C sans résultat à paramètre numéro de signal. On associe un traitant d'interruption à une interruption par **FredSignalAttach**() (p. 41). Par défaut à l'initialisation(**FredSignalsInit**() (p. 41)), on installe pour les interruptions un traitant par défaut : **defaultFredSignalHandler**() (p. 40) .

Pour voir la liste des signaux UNIX : "man -S 7 signal"

1.3.4 Inhibition des interruptions

Sur un monoprocesseur interruptible par des évènements internes ou externes, le mécanisme de base pour assurer l'exclusion mutuelle est d'inhiber temporairement les interruptions. Dans FRED, on gardera cependant la possibilite de tuer le process UNIX (par Ctrl_C) même en cas de programme en boucle infinie en exclusion mutuelle donc masquée.

FRED n'offre qu'un simple mécanisme de masquage/démasquage des interruptions utilisable via les 2 procédures **FredMaskOn**() (p. 40) et **FredMaskOff**() (p. 39). Les masques utilisés sont initialisés par **FredMasksInit**() (p. 40).

Avertissement :

Il est incorrect d'imbriquer les sections critiques.

```
/* code incorrect */
....FredMaskOn();....; FredMaskOn();....FredMaskOff();....FredMaskOff();....
```

1.3.5 Gestion du décompteur

Le plus petit mécanisme de base pour construire une gestion en temps partagé, un échéancier ou la gestion de la date est le compteur à rebours ou décompteur. Un compteur initialement chargé avec une valeur positive est décrémenté périodiquement. Au passage à zéro, une interruption est engendrée et le décomptage s'arrête. FRED offre un décompteur unique pour la gestion du temps partagé. Il est basé sur l'utilisation d'un des compteurs UNIX : le timer virtuel (ITIMER_VIRTUAL) qui ne décompte que durant l'exécution du processus qui l'utilise. Il est utilisé en décompteur. L'interruption qu'il engrendre est le signal SIGVTALRM.

Il est alors possible de lancer un décomptage en microsecondes(**FredCountDownStart**() (p. 42)) ou de l'arrêter (**FredCountDownStop**() (p. 43)). Dans les deux cas, on récupère la valeur résiduelle du décompte en cours au moment de ces invocations. On peut directement lire la valeur résiduelle par **FredCountDownGet**() (p. 42) . Par **FredCountDownAttach**() (p. 42) , on peut attacher un traitant d'interruption à la fin décomptage.

1.3.6 E/S et gestion de mémoire "thread-safe"

Pour rendre **thread safe** , les procédures de la bibliothèque standard C, il suffit de les mettre en exclusion mutuelle. Dans le cadre de FRED, cela est fait par inhibition des interruptions(cf. **Interface de gestion mémoire** (p. 22) et **Interface d'entrée/Sortie** (p. 24))

1.3.7 Gestion des threads

Une fois la commutation de contexte établie, la gestion des threads est relativement simple. Le cycle de vie d'un thread est le suivant :
- A sa création par **FredThreadSpawn**() (p. 17) , il lui est alloué un descripteur **FredThreadDesc** (p. 50), de la mémoire pour sa pile et une description de l'appel de procédure initiale. Après initialisation de son contexte, il est **prêt** à s'exécuter. On le met dans la file d'attente des threads prêts **readyQ** (p. 48) .
- Lorsque c'est son tour d'avoir le processeur selon la règle du tourniquet (temps partagé équitable), il devient actif. C'est le thread **courant**. Il le reste pour une durée bornée (**quantum** de temps). On démarre donc le décompteur lorsqu'il est activé.
- Le thread reste actif soit jusqu'à l'interruption de fin de décomptage (exécution de **QuantumHandler**() (p. 47)) qui le recycle en fin de file des prêts pour relancer un autre thread. Si le thread se bloque avant la fin de quantum dans une file d'attente de synchronisation (**FredThreadSuspend**() (p. 21)), on arrête le décomptage et on active un autre thread prêt (en relançant le décomptage).
- Lorsque le thread est réveillé par un autre thread **FredThreadWakeUp**() (p. 21) , il est remis en queue de file des prêts.
- Un thread actif peut aussi se terminer normalement à la fin de sa procédure initiale ou par **FredThreadExit**() (p. 16). La terminaison du thread implique de libérer sa pile. Un autre thread prêt est alors activé.

L'initialisation de la pile du thread et de son contexte nécessite de déterminer la taille minimum nécessaire et le sens de croissance de la pile c'est à dire vers les adresses faibles ou l'inverse. C'est le rôle du programme **inspect_stack.c** de déterminer cela et de générer le fichier **Cstack.h** qui contiendra ces informations nécessaires à la création d'un thread.

La libération de la mémoire contenant pile par une procédure s'exécutant dans cette pile engendre des erreurs. Il est nécessaire d'exécuter ce code de libération dans une pile différente. FRED utilise pour cela un thread spécialisé dans le nettoyage de la mémoire **leon** (p. 48) qui est créé au départ du programme concurrent. Les threads qui se terminent s'inscrivent dans une file de décédés **deadQ** (p. 48) et réveillent leon.

Le moteur du noyau de thread est donc le mécanisme qui relance systématiquement le thread en tête de la file des prêts **readyQ** (p. 48). Si la file est vide, le moteur est en panne. Pour éviter cette situation, on introduit un thread qui ne se termine jamais et ne fait que passer la main aux autres threads. Ce chien de garde **watchdog** (p. 48) est démarré au lancement de FRED (**FredInitialize**() (p. 14))

Le programme FRED démarre comme un programme C standard. FRED est disponible après son initialisation **FredInitialize**() (p. 14). Pour que l'on puisse utiliser toute l'API FRED dans la procédure **main**(), il est nécessaire de lui adjoindre un thread **mainThread** (p. 62) que l'on pourra interrompre et synchroniser comme tout autre thread.

Enfin, la gestion de la terminaison nécessite la mise en place d'un protocole entre **leon** (p. 48) qui détruit effectivement les threads et **FredWaitTerminate**() (p. 14) qui permet au thread principal d'attendre la fin des autres threads (sauf **leon** (p. 48) et watcdog).

Pour plus de détails, voir **Interface de gestion de threads** (p. 15) et **Implantation des threads** (p. 43)

2 Index des modules

2.1 Modules

Liste de tous les modules

3 Index des composants

3.1 Liste des composants

Liste des classes, des strutures et des unions avec une brève description :

4 Index des fichiers

4.1 Liste des fichiers

Liste de tous les fichiers documentés avec une brève description :

5 Documentation du module

5.1 Interface FRED

Ce module définit l'interface applicative de FRED.

Modules

- **Interface d'initialisation & terminaison**
  Ce module définit comment initier et terminer un programme concurrent FRED.
- **Interface de gestion de threads**
  Ce module définit comment créer des threads, terminer un thread ou un programme concurrent et connaitre le thread courant.
- **Interface d'exclusion mutuelle**
  Ce module définit comment exécuter une séquence de code en exclusion mutuelle.
- **Interface de gestion de file d'attente**
  Ce module définit comment bloquer ou réveiller un thread dans une file d'attente.
- **Interface de gestion mémoire**
  Ce module définit comment allouer ou libérer de la mémoire atomiquement.
- **Interface d'entrée/Sortie**
  Ce module définit comment faire des E/S atomiquement.

5.1.1 Description détaillée

5.2 Interface d'initialisation & terminaison

Ce module définit comment initier et terminer un programme concurrent FRED.

Fonctions

- void **FredInitialize** ()
  Initialisation de FRED.
- void **FredWaitTerminate** ()
  Attente de fin d'exécution de tous les threads créés.

– void **FredAbort** (char ∗mess)
  Arrêt brutal du programme et impression d'un message d'erreur.

5.2.1 Description détaillée

Pour savoir comment l'utiliser, voir **Initialisation & terminaison** (p. 2) et **Les N philosophes** (p. 6)

5.2.2 Documentation de la fonction

5.2.2.1 void FredAbort (char ∗ *mess*)

Paramètres :
  mess message d'erreur fatale

Implantation

```
{
  printf(" \n FATAL  ERROR : ");
  printf(mess);
  printf("\n");
  exit(1); /* fin du programme */
}
```

5.2.2.2 void FredInitialize ()

Implantation La procédure principale **main**() initialise les variables d'état des différents modules de base, se transforme en un thread principal et lance le nettoyeur leon et le veilleur watchdog.

```
{
  FredMasksInit(); /* initialiser les masques de signaux */
  FredSignalsInit();/* initialiser les traitants de signaux */
  nbFredThreads=0;/* aucun thread n'existe */
  FredQueueInit(&readyQ); /* file vide */
  TimeSharingInit(250); /* initialiser le temps partagé */
  MainThreadInit(); /* forger un pseudo thread principal */
  LeonInit(); /* créer le nettoyeur */
  WatchDogInit(); /* créer le chien de garde */
}
```

5.2.2.3 void FredWaitTerminate ()

Implantation Le thread principal se bloque s'il y d'autres threads que le watchdog et leon. Sinon il termine.

```
{ int nonfini=1;
  while(nonfini){
    FredMaskOn();/* début d'exclusion mutuelle */
```

```
    /* Il y a 2 threads de servitude : leon et watchdog */
    if(nbFredThreads>2) /* il y a d'autres threads */
      /* suspension dans la file d'attente de fin */
      FredThreadSuspend(&endQ);
    else  /* il n'y a plus que  leon et watchdog */
      nonfini=0;
    FredMaskOff();/* fin d'exclusion mutuelle */
  }
}
```

5.3 Interface de gestion de threads

Ce module définit comment créer des threads, terminer un thread ou un programme concurrent et connaitre le thread courant.

Définitions des macros

– #define **FredTHREAD_NIL** (**FredThread**)NIL
  Définition du thread inexistant.

Définitions des types

– typedef void ∗(∗ **FredRunnable**)(void ∗)
  Le type d'une procédure initiale de thread.

– typedef **FredThreadDesc** ∗ **FredThread**
  L'identité d'un thread est un pointeur sur son descripteur.

– typedef **FredList** **FredQueue**
  Définition du type queue de threads.

Fonctions

– **FredThread** **FredThreadSpawn** (int stacksz, **FredRunnable** function, void ∗arg, void ∗∗resref)
  Création et lancement d'un thread.

– void **FredThreadExit** (void ∗res)
  Terminaison immédiate du thread courant.

– void **FredThreadYield** ()
  Le thread courant passe la main à un autre thread.

– **FredThread** **FredThreadCurrent** ()
  Rend l'identité du thread courant.

5.3.1 Description détaillée

Pour savoir comment l'utiliser, voir **Gestion des threads** (p. 3) et **Les N philosophes** (p. 6)

5.3.2 Documentation de la macro

5.3.2.1 #define FredTHREAD_NIL (FredThread)NIL

Implantation

C'est un simple alias du pointeur nul de cellule de liste

5.3.3 Documentation du type

5.3.3.1 typedef FredList FredQueue

Implantation

C'est un simple alias d'une liste.

5.3.4 Documentation de la fonction

5.3.4.1 FredThread FredThreadCurrent ()

Renvoie :
l'adresse du descripteur du thread actif

Implantation

```
{
  return current;
}
```

5.3.4.2 void FredThreadExit (void * *res*)

Paramètres :
res adresse du résultat du calcul du thread

Implantation

On rend le résultat à l'endrioit défini au mau moment de la création du thread par FreedThreadSpawn() . Le thread est mis dans la file des décédés pour que sa mémoire soit libérer par **leon** (p. 48)

```
{
  FredThread old;
  FredCountDownStop();/* arrêt décomptage */
  /* mettre le résultat à l'endroit demandé */
  if(current->resref)*(current->resref)=res;
  FredMaskOn();/* début d'exclusion mutuelle */
```

```
  /* réveil de leon */
  FredThreadWakeUp(&leonQ);
  /*
    un thread ne peut libérer la mémoire
    dans laquelle il s'exécute. Il délègue cela
    a leon  en s'inscrivant dans la
    file des threads défunts
  */
  FredThreadSuspend(&deadQ);
  /* on ne revient jamais ici */
  FredAbort(" erreur fatale dans threadDie");
}
```

5.3.4.3 FredThread FredThreadSpawn (int *stacksz*, FredRunnable *function*, void ∗ *arg*, void ∗∗ *resref*)

Paramètres :

stacksz taille de pile exigée pour le thread

function fonction à exécuter par le thread

arg adresse de l'argument de la fonction

resref Si l'adresse resref est non nulle, on y mettra l'adresse du résultat de la fonction lorsque le thread créé terminera.

Renvoie :

l'adresse du descripteur du thread créé ou FredTHREAD_NIL (0)

Implantation

On alloue la mémoire pour un descripteur de thread et la pile (éventuellement arrondi). Si cette étape est franchi avec succès, on initialise le descripteur du thread (**FredThreadDesc** (p. 45)) et son contexte. On l'insère enfin dans la file des prêts **readyQ** (p. 48) en incrémentant le nombre de threads vivants **nbFredThreads** (p. 48) .

```
{
  FredThread nw;
  /* allocation d'un descripteur de thread si possible */
  nw= FredNEW(FredThreadDesc) ;
  if(!nw) return nw;
  /* calcul taille de pile */
  nw->stacksz=FredSTACK_SIZE(stacksz);
  /* allocation de la pile  si possible */
  if(!(nw->stack=(char*)FredAtomicAlloc(nw->stacksz))) {
    /* plus de mémoire */
    FredDELETE( nw); return FredTHREAD_NIL;
  }
  /* initialisation du descripteur */
  nw->err=0; /* init errno à OK */
  nw-> entry=function; /* fonction à exécuter */
  nw->val=arg; /* son argument */
  nw->resref=resref; /* où mettre le résultat */
  /* initialisation du contexte processeur du thread */
  FredContextInit(nw->ctx,
      FredMAKE_SP(nw->stack,nw->stacksz),
```

```
      FredThreadHarness);
  FredMaskOn();/* début d'exclusion mutuelle */
  FredQueuePutLast(&readyQ,nw);/* réveil du nouveau thread */
  nbFredThreads++; /* un thread de plus !! */
  FredMaskOff();/* fin d'exclusion mutuelle */
  return nw;
}
```

5.3.4.4 void FredThreadYield ()

Implantation

```
{
  FredThread old;
  FredMaskOn(); /* début d'exclusion mutuelle */
  current->err=errno; /* sauver errno */
  FredCountDownStop();/* arrêt décomptage */
  old=current;
  /* le courant se met en queue des prêts */
  /* on tire un nouveau courant */
  FredQueuePutLast(&readyQ,current);
  current=FredQueueGetFirst(&readyQ);
  /* traiter le cas d'un unique thread actif : rien faire */
  if(current!=old)FredContextSwitch(old->ctx,current->ctx);
  /*
    on reviendra ici lors de son réveil :
    on relance le décomptage du quantum
  */
  FredCountDownStart(quantum);/* lancer décomptage */
  errno=current->err;/* restaurer errno */
  FredMaskOff();/* fin d'exclusion mutuelle */
}
```

5.4 Interface d'exclusion mutuelle

Ce module définit comment exécuter une séquence de code en exclusion mutuelle.

Définitions des macros

- #define **FredMutexOn**() FredMaskOn()
  Début de section critique.
- #define **FredMutexOff**() FredMaskOff()
  Fin de section critique.

5.4.1 Description détaillée

Cela se fait par simple masquage et démasquage d'interruptions. Pour savoir comment l'utiliser, voir **Exclusion mutuelle** (p. 3) , **Un exemple d'outil de synchronisation : le sémaphore** (p. 5) et **Les N philosophes** (p. 6)

5.4.2 Documentation de la macro

5.4.2.1 #define FredMutexOff() FredMaskOff()

Implantation

C'est un simple alias de l'autorisation des interruptions

5.4.2.2 #define FredMutexOn() FredMaskOn()

Implantation

C'est un simple alias de l'inhibition des interruptions

5.5 Interface de gestion de file d'attente

Ce module définit comment bloquer ou réveiller un thread dans une file d'attente.

Définitions des macros

- #define **FredQueueGetFirst**(Q) (**FredThread**)FredListGetFirst((**FredList**∗)Q)
  Retirer le premier thread d'une file d'attente.
- #define **FredQueuePutLast**(Q, th) FredListPutLast((**FredList**∗)Q,(**FredCell**∗)th)
  Insérer un thread déja bloqué dans une file d'attente.
- #define **FredQueueEmpty**(Q) FredListEmpty((**FredList**∗)Q)
  Tester une queue à vide de threads.
- #define **FredQueueInit**(Q) FredListInit((**FredList**∗)Q) ;
  Initialiser une queue vide de threads.

Fonctions

- void **FredThreadSuspend** (**FredQueue** ∗q)
  Suspendre le thread courant dans une file.
- void **FredThreadWakeUp** (**FredQueue** ∗q)
  Réveille un thread bloqué.

5.5.1 Description détaillée

Pour savoir comment l'utiliser, voir **File d'attente** (p. 4) , **Un exemple d'outil de synchronisation : le sémaphore** (p. 5) et **Les N philosophes** (p. 6)

5.5.2 Documentation de la macro

5.5.2.1 #define FredQueueEmpty(Q) FredListEmpty((FredList*)Q)

Paramètres :
Q queue

Renvoie :
1 si vide. 0 sinon

Implantation

C'est un simple alias du test de liste vide, les conversions de type en sus.

5.5.2.2 #define FredQueueGetFirst(Q) (FredThread)FredListGetFirst((FredList*)Q)

Paramètres :
Q FredQueue où mettre le thread courant

Renvoie :
le premier thread de la file ou 0 si file vide

Avertissement :
Cette procédure n'est utilisable qu'au sein d'une section en exclusion mutuelle.

Implantation

C'est un simple alias du retrait de liste, les conversions de type en sus.

5.5.2.3 #define FredQueueInit(Q) FredListInit((FredList*)Q) ;

Paramètres :
Q queue vide

Implantation

C'est un simple alias de l'initialisation de liste , les conversions de type en sus.

5.5.2.4 #define FredQueuePutLast(Q, th) FredListPutLast((FredList*)Q,(FredCell*)th)

Paramètres :
Q FredQueue où mettre le thread bloqué
th le thread bloqué à insérer en queue

Avertissement :
Cette procédure n'est utilisable qu'au sein d'une section en exclusion mutuelle.

Implantation

C'est un simple alias de l'insertion de liste, les conversions de type en sus.

5.5.3 Documentation de la fonction

5.5.3.1 void FredThreadSuspend (FredQueue ∗ *q*)

Paramètres :

q file où mettre le thread courant

Le thread courant est mis en queue de la file. Un nouveau thread est élu pour s'exécuter.

Avertissement :

Cette procédure n'est utilisable qu'au sein d'une section en exclusion mutuelle.

Implantation

On doit arrêter puis relancer un décomptage.

```
{
 FredThread old;
 current->err=errno; /* sauver errno */
 FredCountDownStop(); /* Arrêt décompeur */
 /* le courant se  en queue de la file q */
 FredQueuePutLast(q,current);
 /*
   on choisit un nouveau courant : on doit garantir
   que la file readyQ n'est jamais vide : C'est le rôle
   du watchdog
 */
 old=current; current=FredQueueGetFirst(&readyQ);
 FredContextSwitch(old->ctx,current->ctx);
 /*
   on reviendra ici lors de son réveil :
   on relance le décomptage du quantum
 */
   FredCountDownStart(quantum);
   errno=current->err;/* restaurer errno */
}
```

5.5.3.2 void FredThreadWakeUp (FredQueue ∗ *q*)

Paramètres :

q queue dont- le premier thread est remis prêt

Le thread réveillé est mis en queue de la file des prêts. C'est une opération nulle si la file est vide.

Avertissement :

Cette procédure n'est utilisable qu'au sein d'une section en exclusion mutuelle.

Implantation

```
{
   if(!FredQueueEmpty(q))FredQueuePutLast(&readyQ,FredQueueGetFirst(q));
}
```

5.6 Interface de gestion mémoire

Ce module définit comment allouer ou libérer de la mémoire atomiquement.

Définitions des macros

- #define **FredNEW**(T) (T*)FredAtomicAlloc(sizeof(T))
  Alloue **atomiquement** *un bloc pour un type donné.*
- #define **FredNEW_VECTOR**(T, CT) (T*)FredAtomicCalloc(sizeof(T),(CT))
  Alloue **atomiquement** *un vecteur de blocs pour un type donné.*
- #define **FredDELETE**(X) FredAtomicFree((void*)X)
  Libère **atomiquement** *un bloc ou un vecteur de blocs précédemment alloué.*

Fonctions

- void * **FredAtomicAlloc** (int x)
  Alloue **atomiquement** *un bloc d'octets.*
- void * **FredAtomicCalloc** (int x, int y)
  Alloue **atomiquement** *un vecteur de blocs d'octets.*
- void **FredAtomicFree** (void *x)
  Libère **atomiquement** *un bloc précédemment alloué.*

5.6.1 Description détaillée

Il suffit d'exécuter les procédure de la bibliothèque C en exclusion mutuelle.

5.6.2 Documentation de la macro

5.6.2.1 #define FredDELETE(X) FredAtomicFree((void*)X)

Paramètres :
X bloc à libérer L'intérêt est de dissimuler la conversion de type

5.6.2.2 #define FredNEW(T) (T*)FredAtomicAlloc(sizeof(T))

Paramètres :
T type à allouer

Renvoie :
un pointeur sur une valeur du type ou 0

L'intérêt est de dissimuler les concersions de type et calcul de taille en octets du bloc à allouer.

5.6.2.3 #define FredNEW_VECTOR(T, CT) (T*)FredAtomicCalloc(sizeof(T),(CT))

Paramètres :

- ***T*** type à allouer
- ***CT*** nombre d'item dans le vecteur

Renvoie :

un pointeur sur un vecteur du type ou 0

L'intérêt est de dissimuler les conversions de type et calcul de taille en octets du bloc à allouer.

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

5.6.3 Documentation de la fonction

5.6.3.1 void* FredAtomicAlloc (int *x*)

Paramètres :

- ***x*** nombre d'octets à allouer

Renvoie :

un pointeur sur un bloc ou 0

Voir le manuel unix pour malloc

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  FredMaskOn();
  void * y=malloc(x);
  FredMaskOff();
  return y;
};
```

5.6.3.2 void* FredAtomicCalloc (int *x*, int *y*)

Paramètres :

- ***x*** nombre d'octets à allouer par bloc
- ***y*** nombre de blocs

Renvoie :

un pointeur sur un vecteur de blocs ou 0

Voir le manuel unix pour calloc

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  FredMaskOn();
  void * z=calloc(x,y);
  FredMaskOff();
  return z;
};
```

5.6.3.3 void FredAtomicFree (void ∗ *x*)

Paramètres :
  x bloc à libérer

Voir le manuel unix pour free

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  FredMaskOn();free(x);FredMaskOff();
};
```

5.7 Interface d'entrée/Sortie

Ce module définit comment faire des E/S atomiquement.

Fonctions

- int **FredPrintf** (const char ∗format,...)
  Imprime **atomiquement** *sur la sortie standard.*
- int **FredFprintf** (FILE ∗stream, const char ∗format,...)
  Imprime **atomiquement** *dans un fichier.*
- int **FredSprintf** (char ∗str, const char ∗format,...)
  Imprime **atomiquement** *dans un buffer.*
- int **FredScanf** (const char ∗format,...)
  lit et décompose **atomiquement** *un texte depuis un fichier source*
- int **FredFscanf** (FILE ∗stream, const char ∗format,...)
  lit et décompose **atomiquement** *un texte depuis l'entrée standard*
- int **FredSscanf** (char ∗str, const char ∗format,...)
  lit et décompose **atomiquement** *un texte depuis un buffer*

5.7.1 Description détaillée

Il suffit d'exécuter les procédure de la bibliothèque C en exclusion mutuelle.

5.7.2 Documentation de la fonction

5.7.2.1 int FredFprintf (FILE ∗ *stream*, const char ∗ *format*, ...)

Paramètres :

stream le fichier cible

format format d'impression

... suite variable d'argument

Renvoie :

un code d'erreur

Voir le manuel unix pour fprintf

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  int y;va_list ap;
  FredMaskOn();
  y=vfprintf(stream,format,ap);
  FredMaskOff();
  return y;
};
```

5.7.2.2 int FredFscanf (FILE ∗ *stream*, const char ∗ *format*, ...)

Paramètres :

stream le fichier source

format format d'analyse

... suite variable d'argument

Renvoie :

un code d'erreur

Voir le manuel unix pour fscanf

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  int y;va_list ap;
  FredMaskOn();
  y=vfscanf(stream,format,ap);
  FredMaskOff();
  return y;
};
```

5.7.2.3 int FredPrintf (const char ∗ *format*, ...)

Paramètres :

format format d'impression

... suite variable d'argument

Renvoie :

un code d'erreur

Voir le manuel unix pour printf

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  int y;
  va_list ap;
  va_start(ap,format);
  FredMaskOn();
  y=vprintf(format,ap);
  FredMaskOff();
  return y;
};
```

5.7.2.4 int FredScanf (const char ∗ *format*, ...)

Paramètres :

format format d'analyse

... suite variable d'argument

Renvoie :

un code d'erreur

Voir le manuel unix pour scanf **Implantation** Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  int y;
  va_list ap;
  va_start(ap,format);
  FredMaskOn();
  y=vscanf(format,ap);
  FredMaskOff();
  return y;
};
```

5.7.2.5 int FredSprintf (char ∗ *str*, const char ∗ *format*, ...)

Paramètres :

str zone pour recevoir la chaine de caractère formatée

format format d'impression

... suite variable d'argument

Renvoie :
un code d'erreur

Voir le manuel unix pour sprintf

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  int y;va_list ap;
  FredMaskOn();
  y=vsprintf(str,format,ap);
  FredMaskOff();
  return y;
};
```

5.7.2.6 int FredSscanf (char ∗ *str*, const char ∗ *format*, ...)

Paramètres :
str le buffer source

format format d'analyse

... suite variable d'argument

Renvoie :
un code d'erreur

Voir le manuel unix pour sscanf

Implantation Simple mise en exclusion mutuelle de la procédure équivalente de la lib C.

```
{
  int y;va_list ap;
  FredMaskOn();
  y=vsscanf(str,format,ap);
  FredMaskOff();
  return y;
};
```

5.8 Interface sémaphore

Une implémentation classique du classique sémaphore de Dijkstra.

Composants

– struct **FredSemaphore**
Descripteur du sémaphore.

Définitions des types

– typedef FredSemaphore **FredSemaphore**
  Descripteur du sémaphore.

Fonctions

– void **FredSemaphoreInit** (**FredSemaphore** ∗s, int i)
  Initialise le descripteur de sémaphore.

– void **FredSemaphoreP** (**FredSemaphore** ∗s)
  Prend un jeton.

– void **FredSemaphoreV** (**FredSemaphore** ∗s)
  Depose un jeton.

5.8.1 Description détaillée

5.8.2 Documentation du type

5.8.2.1 typedef struct FredSemaphore FredSemaphore

Un sémaphore est décrit par un compteur et une file d'attente. La valeur initiale du compteur doit être >= 0

Implantation Le compteur représente le nombre de jetons si le compteur est >=0 et le nombre de threads en attente s'il est <0.

5.8.3 Documentation de la fonction

5.8.3.1 void FredSemaphoreInit (FredSemaphore ∗ *s*, int *i*)

Paramètres :

s adresse descripteur sémaphore

i nombre initial de jetons

Avertissement :

la valeur initiale du sémaphore doit être positive ou nulle

```
{
  FredMutexOn();/* début d'exclusion mutuelle */
  /* le semaphore existe et la valeur initiale est >=0 */
  if ((s !=(FredSemaphore *) 0)&&(i>=0)){
    s->counter = i;
    FredQueueInit(&(s->queue));/* file vide */
  }
  FredMutexOff();/* fin d'exclusion mutuelle */
}
```

5.8.3.2 void FredSemaphoreP (FredSemaphore ∗ *s*)

Paramètres :
s adresse descripteur sémaphore

S'il n'y a pas de jeton, le thread se bloque. Il sera réveillé à son tour (file d'attente FIFO) quand un jeton lui sera donné.

Implantation

```
{
  FredMutexOn();/* début d'exclusion mutuelle */
  s->counter--;
  if(s->counter < 0){
    /* pas de jeton :  on suspend le thread actif   */
    FredThreadSuspend(&(s->queue))  ;
  }
  FredMutexOff();/* fin d'exclusion mutuelle */
}
```

5.8.3.3 void FredSemaphoreV (FredSemaphore ∗ *s*)

Paramètres :
s adresse descripteur sémaphore

S'il y a des threads en attente d'un jeton, on réveille le + ancien.

Implantation

```
{
  FredMutexOn();/* début d'exclusion mutuelle */
  s->counter++;
  if(s->counter <=0){
    /* il y a des demandeurs en attente */
    FredThreadWakeUp(&(s->queue));
  }
  FredMutexOff();/* fin d'exclusion mutuelle */
}
```

5.9 Implémentation de FRED

Ce module regroupe les sous modules nécessaires à la mise en oeuvre de FRED.

Modules

– **Interface de gestion de listes**
Ce module est une gestion de liste FIFO.

– **Interface de gestion de contexte**
Ce module regroupe plusieurs implantations de la commutation de contexte processeur.

– **Interface de masquage/démasquage des interruptions**

- **Interface de traitement des interruptions**
- **Interface de gestion du décompteur**
- **Implantation des threads**

5.9.1 Description détaillée

5.10 Interface de gestion de listes

Ce module est une gestion de liste FIFO.

Composants

- struct **FredCell**
  descripteur de cellule de liste
- struct **FredList**
  descripteur de liste

Définitions des macros

- #define **NIL** ((**FredCell**∗)0)
  macro pour définir le pointeur vide
- #define **FredCELL** **FredCell** _cell
  macro pour définir une structure comme une cellule de liste
- #define **FredListEmpty**(Q) (((Q) → last) == (**FredCell**∗)0)
  Test de file vide.

Définitions des types

- typedef FredCell **FredCell**
  descripteur de cellule de liste
- typedef FredList **FredList**
  descripteur de liste

Fonctions

- void **FredListPutLast** (**FredList** ∗q, **FredCell** ∗b)
  Insertion en queue d'une cellule.
- **FredCell** ∗ **FredListGetFirst** (**FredList** ∗q)
  Retrait de la cellule de tête.
- void **FredListInit** (**FredList** ∗q)
  Initialisation de la liste.

5.10.1 Description détaillée

Elle est implantée comme une liste circulaire doublement chaînée avec insertion en queue et retrait en tête.

5.10.2 Documentation de la macro

5.10.2.1 #define FredCELL FredCell _cell

Pour définir une structure qyelconque comme pouvant être insérer dans une liste, il suffit d'insérer FredCELL en premiere définition des composants de la structure

```
struct MyCell{
  FredCELL;
  int x;
  int y;
  ...
};
```

5.10.2.2 #define FredListEmpty(Q) (((Q) → last) == (FredCell*)0)

5.10.2.3 #define NIL ((FredCell*)0)

5.10.3 Documentation du type

5.10.3.1 typedef struct FredCell FredCell

Dans une liste circulaire à 1 élément, celui ci est son propre successeur et prédécesseur. Il est donc doublement rechainé sur lui-même.

5.10.3.2 typedef struct FredList FredList

Une liste circulaire pointe sur le dernier élément. Pour une liste vide, ce derrnier élément est nul.

5.10.4 Documentation de la fonction

5.10.4.1 FredCell* FredListGetFirst (FredList * *q*)

Paramètres :
 q liste où retirer

Renvoie :
 cellule retirer **Implantation**

Le retrait en tête nécessite de traiter le cas d'une file vide et le cas d'une file à 1 élément. Le cas d'une file à 2 cellules qui devient à 1 cellule(qui est chainée sur elle même) peut se traiter comme le cas général.

```
{
  FredCell* b;
  if(FredListEmpty(q)){
    /* file vide */
    return NIL;
  }else if ( q->last->next==q->last){
    /* file à 1 item */
    b=q->last;
    q->last=NIL; /* la file devient vide */
  }else {
    b=q->last->next;
    b->pred->next=b->next;
    b->next->pred=b->pred;
  }
  b->next=b->pred=NIL; /* la cellule n'est plus chainée */
  return b;
}
```

5.10.4.2 void FredListInit (FredList ∗ *q*)

Paramètres :

q liste à initialiser

Implantation

```
{
  q->last=NIL;
}
```

5.10.4.3 void FredListPutLast (FredList ∗ *q*, FredCell ∗ *b*)

Paramètres :

q liste où insérer

b cellule à insérer

Implantation

L'insertion en queue place la nouvelle cellule entre le dernier et le premier. Elle devient le nouveau dernier item. Deux cas particuliers sont la file vide et la file à une cellule (qui est chainée sur elle même). Ce dernier cas peut se programmer comme le cas général.

```
{
  if(FredListEmpty(q)){
    /* file vide */
    b->next=b->pred=b;
    q->last=b;
  }else{
```

```
    b->next=q->last->next;
    b->pred=q->last;
    b->next->pred=b;
    b->pred->next=b;
    q->last=b;
  }
}
```

5.11 Interface de gestion de contexte

Ce module regroupe plusieurs implantations de la commutation de contexte processeur.

Modules

- **Gestion de contexte pour Linux IA32 par setjmp/longjmp**
- **Gestion de contexte pour Linux IA32 (code natif)**
- **Gestion de contexte pour Solaris Sparc32 par setjmp/longjmp**

5.11.1 Description détaillée

5.12 Gestion de contexte pour Linux IA32 par setjmp/longjmp

Définitions des types

- typedef jmp_buf **LinuxIA32FredContext**
  Contexte de sauvegarde : un jmp_buf de linux.

Fonctions

- void **LinuxIA32FredContextRestore** (**LinuxIA32FredContext** ctx)
  Restaurer un contexte processeur. Le contexte courant est perdu.
- void **LinuxIA32FredContextSwitch** (**LinuxIA32FredContext** old, **LinuxIA32FredContext** ctx)
  Restaurer un contexte processeur. Le contexte courant est sauvé.
- void **LinuxIA32FredContextInit** (**LinuxIA32FredContext** ctx, char ∗sp, void(∗entry)())
  Initialiser un contexte processeur.

5.12.1 Documentation de la fonction

5.12.1.1 void LinuxIA32FredContextInit (LinuxIA32FredContext *ctx*, char ∗ *sp*, void(∗ *entry*)())

Paramètres :

ctx contexte à initialiser

sp sommet de pile

entry adresse de la procédure à démarrer

On initialise un pointeur de pile et une adresse d'instruction.

Implantation

C'est totalement dépendant du type de processeur et de l'implantation des fonction setjmp() et longjmp() car on doit connaitre les registres à modifier et leurs emplacements dans le jmp_buf.

```
{
  ctx->__jmpbuf[JB_SP] = (int)sp;
  ctx->__jmpbuf[JB_PC] = (int)entry;
};
```

5.12.1.2 void LinuxIA32FredContextRestore (LinuxIA32FredContext *ctx*)

Paramètres :

ctx contexte à restaurer

C'est l'équivalent d'un **goto** sans espoir de reprise. **Implantation**

C'est exactement un `longjmp()` de la libC

```
{
  longjmp(ctx,1);
  /*  Il n'y a pas de retour possible en séquence d'un tel appel  */
  KernelAbort(" FredContext Restore  : erreur fatale..... ");
}
```

5.12.1.3 void LinuxIA32FredContextSwitch (LinuxIA32FredContext *old*, LinuxIA32FredContext *ctx*)

Paramètres :

old contexte où sauvegarder

ctx contexte à restaurer

Implantation

C'est exactement une combinaison de `setjmp()` pour la sauvegarde et `longjmp()` pour la restauration

```
{
  if(setjmp(old))return; /* cas de reprise du contexte*/
  /* contexte courant sauvegardé : on restaure un autre contexte*/
  longjmp(ctx,1);
  /* on ne revient jamais ici */
  KernelError(" FredContext switch  : erreur ????...???... ");
}
```

5.13 Gestion de contexte pour Linux IA32 (code natif)

Définitions des types

- typedef int **IA32FredContext** [1]
  Contexte de sauvegarde : un pointeur de sommet de pile où on été empilés tous les autres registes.

Fonctions

- void **IA32FredContextRestore** (**IA32FredContext** ctx)
  Restaurer un contexte processeur. Le contexte courant est perdu.
- void **IA32FredContextSwitch** (**IA32FredContext** old, **IA32FredContext** ctx)
  Restaurer un contexte processeur. Le contexte courant est sauvé.
- void **IA32FredContextInit** (**IA32FredContext** ctx, char ∗sp, void(∗entry)())
  Initialiser un contexte processeur.

5.13.1 Documentation de la fonction

5.13.1.1 void IA32FredContextInit (IA32FredContext *ctx*, char ∗ *sp*, void(∗ *entry*)())

Paramètres :

ctx contexte à initialiser

sp sommet de pile

entry adresse de la procédure à démarrer

On initialise un registre pointeur de pile et un registre adresse d'instruction.

Implantation

C'est totalement dépendant du type de processeur et de la façon dont le code est généré par le compilateur C (ici gcc 3.2) pour passer les paramètres dans la pile et entrer/sortir d'une procédure. Une fois ceci connu (en deassemblant du code généré par ex.), il suffit de décider de l'ordre de sauvegarde des registres.

Ici, seuls l'initialisation du la sauvegarde des registres pointeur de pile et compteur ordinal est nécessaire.

```
{
  /* sauve eax ebx et ecx  pour pouvoir les utiliser*/
  asm("pushl %eax;pushl %ebx; pushl %ecx");
  /* sauve esp  dans ecx */
  asm(" movl %esp, %ecx");
  /* faire pointer eax sur ctx (déplacement 16 relativement à ecx)*/
  asm("  movl 16(%ecx), %eax");
  /* esp pointe sur nouvelle pile sp  (deplacement 20 relativement à ecx)*/
```

```
  asm("  movl 20(%ecx), %esp");
  /* ebx pointe sur la procedure initiale entry (deplacement 24 relativement à ecx)*/
  asm("  movl 24(%ecx), %ebx");
  /* simulation sauvegarde registres    */
  asm(" pushl %ebx "); /* point de démarrage */
  asm(" pushl $0 "); /* ebp bidon */
  asm(" pushl $0 "); /* eax  bidon */
  asm(" pushl $0 "); /* ebx  bidon */
  asm(" pushl $0 "); /* ecx  bidon */
  asm(" pushl $0 "); /* edx  bidon */
  asm(" pushl $0 "); /* edi  bidon */
  asm(" pushl $0 "); /* esi  bidon */
  /* sauver pointeur de pile nouveau dans ctx */
  asm(" movl %esp, 0(%eax)");
  /* restaure esp courant depuis ecx */
  asm(" movl %ecx ,%esp");
  /* restaure  eax ebx et ecx */
  asm("popl %ecx;popl %ebx;popl %eax");
  /* retour */
  asm(" ret");
};
```

5.13.1.2 void IA32FredContextRestore (IA32FredContext *ctx*)

Paramètres :

ctx contexte à restaurer

C'est l'équivalent d'un **goto** sans espoir de reprise.

Implantation

C'est totalement dépendant du type de processeur et de la façon dont le code est généré par le compilateur C (ici gcc 3.2) pour passer les paramètres dans la pile et entrer/sortir d'une procédure. Une fois ceci connu (en deassemblant du code généré par ex.), il suffit de décider de l'ordre de sauvegarde des registres.

Ici on dépile les registres de la pile dont le sommet est obtenu à partir du contexte.

```
{
  /* esp pointeur sur le contexte ctx deplacement 4 relatif à esp*/
  asm( "movl 4(%esp), %esp");
  /* esp pointeur sur la pile à restaurer */
  asm( "movl 0(%esp), %esp");
  /*----- restaurer autres registres  ----- */
  asm( "popl %esi " );
  asm( "popl %edi " );
  asm( "popl %edx " );
  asm( "popl %ecx " );
  asm( "popl %ebx " );
  asm( "popl %eax " );
  asm( "popl %ebp " );
  asm( "ret" );
}
```

5.13.1.3 void IA32FredContextSwitch (IA32FredContext *old*, IA32FredContext *ctx*)

Paramètres :

old contexte où sauvegarder

ctx contexte à restaurer

Implantation

C'est totalement dépendant du type de processeur et de la façon dont le code est généré par le compilateur C (ici gcc 3.2) pour passer les paramètres dans la pile et entrer/sortir d'une procédure. Une fois ceci connu (en deassemblant du code généré par ex.), il suffit de décider de l'ordre de sauvegarde des registres.

Ici on empile les registres sur la pile, sauve le pointeur de pile dans le contexte old et on fait l'opération inverse depuis le contexte ctx.

```
{
  /*  ebp et eax sauvé sur  sur la pile */
  asm( "pushl %ebp  ");
  asm( "pushl %eax  ");
  /*ebp pointeur sur le contexte old  déplacement 12 relatif à  esp */
   asm(" movl 12(%esp), %ebp");
  /*eax pointeur sur le contexte new  déplacement 16 relatif à  esp*/
   asm(" movl 16(%esp), %eax");
  /*----- sauvver autres registres  ----- */
  asm( "pushl %ebx ");
  asm( "pushl %ecx ");
  asm( "pushl %edx ");
  asm( "pushl %edi ");
  asm( "pushl %esi ");
  /* sauver esp dans le contexte old  */
  asm( "movl %esp ,0(%ebp)" );
  /* restaurer  nouvelle pile depuis  depuis eax (new) */
  asm( "movl 0(%eax),  %esp " );
  /*----- restaurer autres registres  ----- */
  asm( "popl %esi  ");
  asm( "popl  %edi ");
  asm( "popl %edx  ");
  asm( "popl  %ecx ");
  asm( "popl  %ebx ");
  asm( "popl  %eax ");
  asm( "popl  %ebp ");
  asm( "ret" );
}
```

5.14 Gestion de contexte pour Solaris Sparc32 par setjmp/longjmp

Définitions des types

– typedef jmp_buf **SolarisSparc32FredContext**

Contexte de sauvegarde : un jmp_buf de solaris.

Fonctions

- void **SolarisSparc32FredContextRestore** (**SolarisSparc32FredContext** ctx)
  Restaurer un contexte processeur. Le contexte courant est perdu.
- void **SolarisSparc32FredContextSwitch** (**SolarisSparc32FredContext** old, **SolarisSparc32FredContext** ctx)
  Restaurer un contexte processeur. Le contexte courant est sauvé.
- void **SolarisSparc32FredContextInit** (FredContext ctx, char ∗sp, void(∗entry)())
  Initialiser un contexte processeur.

5.14.1 Documentation de la fonction

5.14.1.1 void SolarisSparc32FredContextInit (FredContext *ctx*, char ∗ *sp*, void(∗ *entry*)())

Paramètres :

ctx contexte à initialiser

sp sommet de pile

entry adresse de la procédure à démarrer

On initialise un pointeur de pile et une adresse d'instruction.

Implantation

C'est totalement dépendant du type de processeur et de l'implantation des fonction setjmp() et longjmp() car on doit connaitre les registres à modifier et leurs emplacements dans le jmp_buf.

```
{
  ctx[ 1 ]= (long)sp;
  ctx[ 2 ]= (long)entry;
};
```

5.14.1.2 void SolarisSparc32FredContextRestore (SolarisSparc32FredContext *ctx*)

Paramètres :

ctx contexte à restaurer

C'est l'équivalent d'un **goto** sans espoir de reprise.

Implantation

C'est exactement un `longjmp()` de la libC

```
{
  longjmp(ctx,1);
```

```
/*  Il n'y a pas de retour possible en séquence d'un tel appel  */
KernelAbort(" FredContext Restore  : erreur fatale..... ");
}
```

5.14.1.3 void SolarisSparc32FredContextSwitch (SolarisSparc32FredContext *old*, SolarisSparc32FredContext *ctx*)

Paramètres :

old contexte où sauvegarder

ctx contexte à restaurer

Implantation

C'est exactement une combinaison de `setjmp()` pour la sauvegarde et `longjmp()` pour la restauration

```
{
  if(setjmp(old))return; /* cas de reprise du contexte*/
  /* contexte courant sauvegardé : on restaure un autre contexte*/
  longjmp(ctx,1);
  /* on ne revient jamais ici */
  KernelError(" FredContext switch  : erreur ????...???... ");
}
```

5.15 Interface de masquage/démasquage des interruptions

Fonctions

- void **FredMaskOn** ()
  Début d'exclusion mutuelle par masquage des interruptions.
- void **FredMaskOff** ()
  Fin d'exclusion mutuelle par démasquage des interruptions.
- void **FredMasksInit** ()
  Initialisation des masques de validation/inhibition des interruptions.

Variables

- sigset_t **ALL_BLOCKED**
  Masque d'inhibition des interruptions.
- sigset_t **NO_BLOCKED**
  Masque d'autorisation des interruptions.

5.15.1 Documentation de la fonction

5.15.1.1 void FredMaskOff ()

Implantation L'implantation UNIX est très simple. Il suffit de changer le masque des signaux à la valeur définissant les signaux à laisser passer.

```
{
  sigprocmask(SIG_SETMASK, &NO_BLOCKED, (sigset_t *)0);
}
```

5.15.1.2 void FredMaskOn ()

Implantation L'implantation UNIX est très simple. Il suffit de changer le masque des signaux à la valeur définissant les signaux à bloquer.

```
{
  sigprocmask(SIG_SETMASK, &ALL_BLOCKED, (sigset_t *)0);
}
```

5.15.1.3 void FredMasksInit ()

Implantation L'implantation UNIX est très simple. Il suffit de définir le vecteur binaire autorisant/interdisant chacune des interruption. On garde la possibilité de tuer le process UNIX en cas de programme en boucle infinie masquée en laissant passer le signal SIGINT

```
{
  sigfillset(&ALL_BLOCKED); /* on masque toutes les interruptions */
  sigdelset (&ALL_BLOCKED,SIGINT ); /* sauf   SIGINT  */
  sigemptyset(&NO_BLOCKED); /* on demasque toutes les interruptions */
}
```

5.16 Interface de traitement des interruptions

Fonctions

– void **FredSignalAttach** (int sig, void(*handler)(int))
  Définir un traitant d'interruption.

– void **FredSignalsInit** ()
  Initialise les traitants par défaut.

– void **defaultFredSignalHandler** (int sig)
  Traitant par défaut des interruptions.

5.16.1 Documentation de la fonction

5.16.1.1 void defaultFredSignalHandler (int *sig*)

Implantation

Simple impression d'un message d'erreur

```
{
  printf("\nFRFED Default Signal Handler : signal %d \n ", sig);
}
```

5.16.1.2 void FredSignalAttach (int *sig*, void(∗ *handler*)(int))

Paramètres :

sig numéro de l'interruptio(signal UNIX)

handler procédure appelée lors de l'interruption

Implantation

C'est un simple carrossage de la procédure sigaction() d'UNIX.

```
{
  struct sigaction siga;
  siga. sa_handler=handler;
  siga. sa_flags=0;
  sigemptyset(&(siga.sa_mask));
  sigaction(sig,&siga, (struct sigaction* )0);
}
```

5.16.1.3 void FredSignalsInit ()

Implantation

On initialise avec un traitant par défaut sauf pour le signal SIGINT pour conserver le comportement standard UNIX du ctrlC

```
{
  int i; int nsig;
  /* calcul du nombre de signaux */
  nsig=sizeof(sigset_t)*8;
  for(i=0;i<nsig;i++){
    if(i!=SIGINT){
      FredSignalAttach(i,defaultFredSignalHandler);
      /* on indique de redémarrer les appels systèmes interrompus */
      siginterrupt(i,1);
   }
  }
}
```

5.17 Interface de gestion du décompteur

Fonctions

- long **FredCountDownStart** (long uS)
  Initier un décomptage.
- long **FredCountDownStop** ()
  Arrêter un décomptage.

- long int **FredCountDownGet** ()
  Lecture du décompteur.
- void **FredCountDownAttach** (void(∗handler)(int))
  Attache un traitant de fin de décomptage.

5.17.1 Documentation de la fonction

5.17.1.1 void FredCountDownAttach (void(∗ *handler*)(int))

Paramètres :
handler code à exécuter sur it timer

Implantation L'implantation UNIX est très simple. Il suffit d'attacher le traitant au signal SIGVTALRM associé compteur ITIMER_VIRTUAL.

```
{
  FredSignalAttach(SIGVTALRM,handler);
}
```

5.17.1.2 long int FredCountDownGet ()

Renvoie :
le temps restant à décompter du décomptage précédent ou 0

Implantation L'implantation UNIX est très simple. Il suffit de lirer le compteur ITIMER_VIRTUAL .

```
{
   struct itimerval old;
  getitimer((int)ITIMER_VIRTUAL,&old);
  /* temps restant à décompter */
  return old.it_value.tv_usec;
}
```

5.17.1.3 long FredCountDownStart (long *uS*)

Paramètres :
uS durée de décomptage en microsecondes

Renvoie :
le temps restant à décompter du décomptage précédent ou 0

Implantation L'implantation UNIX est très simple. Il suffit de charger le compteur ITIMER_VIRTUAL avec la valeur donnée.

```
{
  struct itimerval q;struct itimerval old;
  /* décomptage coup par coup :
```

```
  un décomptage à la fois et arrêt du timer  en fin de décompte */
  q.it_interval.tv_sec=0;
  q.it_interval.tv_usec=0;
  /* duree du decomptage   */
  q.it_value.tv_sec=0;
  q.it_value.tv_usec = uS ;
  setitimer((int)ITIMER_VIRTUAL,&q,&old);
  return old.it_value.tv_usec;
}
```

5.17.1.4 long FredCountDownStop ()

Renvoie :
le temps restant à décompter du décomptage précédent ou 0

Implantation L'implantation UNIX est très simple. Il suffit de charger le compteur ITIMER_VIRTUAL avec la valeur 0 qui l'arrête.

```
{
  struct itimerval q;struct itimerval old;
  /* décomptage coup par coup :
  un décomptage à la fois et arrêt du timer  */
  q.it_interval.tv_sec=0;
  q.it_interval.tv_usec=0;
  /* duree du decomptage   */
  q.it_value.tv_sec=0;
  q.it_value.tv_usec = 0 ;
  setitimer((int)ITIMER_VIRTUAL,&q,&old);
  /* temps restant à décompter */
  return old.it_value.tv_usec;
}
```

5.18 Implantation des threads

Composants

– struct **FredThreadDesc**
Descripteur de threads.

Définitions des types

– typedef FredThreadDesc **FredThreadDesc**
Descripteur de threads.

Fonctions

– void ∗ **leonCode** (void ∗x)
Le code du nettoyeur leon.

– void **LeonInit** ()

Initialisation de leon le nettoyeur.

– void ∗ **watchDogCode** (void ∗x)
Le code du chien de garde watchdog.

– void **WatchDogInit** ()
Initialisation du chien de garde watchdog.

– void **QuantumHandler** (int i)
Traitement d'interruption de fin de quantum.

– void **TimeSharingInit** (long int uS)
Initialisation du temps partagé.

– void **FredThreadHarness** ()
Code initial des threads.

– void **MainThreadInit** ()
Initialisation du pseudo thread principal.

Variables

– int volatile **nbFredThreads**
Compteur du nombre de threads créés non terminés.

– **FredThread current**
Le thread actif.

– **FredQueue readyQ**
File d'attente des threads prêts.

– **FredQueue deadQ**
File d'attente des threads morts dont la pile est à libérer (réservée à **leon** (p. 48) *le nettoyeur) Un thread terminant s'inscrit dans cette file et réveille* **leon** (p. 48) *s'il n'est pas déja réveillé (cf* **leonCode***()* (p. 45) *).*

– **FredThread leon**
Le nettoyeur leon.

– **FredQueue leonQ**
File d'attente de **leon** (p. 48).

– **FredQueue endQ**
File d'attente de terminaison (réservée au **mainThread** (p. 62) *).*

– **FredThreadDesc mainThreadBlock**
Descripeur du thread associé à la procédure **main***().*

– **FredThread watchdog**
Thread chien de garde.

– long int **quantum**
Valeur du quantum en microseconde.

5.18.1 Documentation du type

5.18.1.1 typedef struct FredThreadDesc FredThreadDesc

Le champ **FredThreadDesc : :err** (p. 51) est la sauvegarde de la variable globale errno qui contient le code d'erreur du programme. On la sauve et la restaure à chaque commutation de thread.

5.18.2 Documentation de la fonction

5.18.2.1 void FredThreadHarness ()

Comme on ne peut retourner de façon normale de la procédure initiale d'un thread et comme le retour de celle-ci est le retour standard généré par le compilateur C, on utilise une procédure qui encadre (harnache) le démarrage et la fin d'un thread.

Implantation

Ce harnais appelle la procédure initiale du thread, récupére son résultat et délégue à **leon** (p. 48) la libération de la pile sans oublier les masquages et décomptages nécessaires.

```
{/* on démarre masqué */
  FredThread old;
  /* lancer le décomptage */
  FredCountDownStart(quantum);
  /* démasquage */
  FredMaskOff();
  /* appel de la procédure initiale avec ou sans récupération dec résultat */
  if(current->resref){ /* on attend un résultat */
      *(current->resref)=current->entry(current->val);

  }else{
      current->entry(current->val);
  }
  /* arrêt décomptage */
  FredCountDownStop();
  /* début exclusion mutuelle */
  FredMaskOn();
  /* réveil de leon */
  FredThreadWakeUp(&leonQ);
  /*
    un thread ne peut libérer la mémoire
    dans laquelle il s'exécute. Il délègue cela
    a leon  en s'inscrivant dans la
    file des threads défunts
  */
  FredThreadSuspend(&deadQ);
  /* on ne revient jamais la */
  FredAbort(" erreur fatale dans threadHarness");
}
```

5.18.2.2 void* leonCode (void * x)

Il est en charge de la libération de mémoire des threads morts et de la détection de terminaison.

Implantation

leon attend qu'un thread mourant le réveille dans sa file d'attente spécifique **leonQ** (p. 44) . Il trouve les threads décédés dans la file **deadQ** (p. 48) et libère la mémoire. Il décrémente le compte de threads **nbFredThreads** (p. 48) et détecte la terminaison. Dans ce cas, il réveille le thread principal si nécessaire.

```
{
  FredThread t;
  while(1){
    FredMaskOn();/* début d'exclusion mutuelle */
    /* libérer la mémoire des threads défunts */
    while((t=FredQueueGetFirst(&deadQ))){
      free(t->stack);free(t);
      nbFredThreads--;
    }
    /* si terminaison, on réveille le thread l'attendant */
    if(nbFredThreads==2)FredThreadWakeUp(&endQ);
    /* leon attend de nouveaux décès */
    FredThreadSuspend(&leonQ);
    FredMaskOff();
  }
  /* on ne revient jamais la */
  FredAbort(" erreur fatale dans leon");
}
```

5.18.2.3 void LeonInit ()

Implantation On lance leon après avoir initialiser les 2 files **leonQ** (p. 44) et **deadQ** (p. 48) .

```
{
  FredQueueInit(&deadQ); /*  file vide */
  FredQueueInit(&leonQ); /*  file vide */
  leon=FredThreadSpawn(20000,leonCode,(void*)0,(void**)0);
}
```

5.18.2.4 void MainThreadInit ()

Implantation La procédure principale **main**() initialise la file d'attente de terminaison, son bloc descripteur et devient le thread courant. A partir de la, elle peut utiliser l'interface FRED comme tout thread normal (synchronisation etc..). Elle peut attendre la terminaison des autres threads via la file **endQ** (p. 44) (cf. **FredWaitTerminate**() (p. 14)).

```
                     {
  /* file d'attente de disparition des autres threads */
  FredQueueInit(&endQ); /*  file vide */
  /* allocation decripteur */
  mainThread=&mainThreadBlock;
```

```
  /* thread actif */
  current=mainThread;
}
```

5.18.2.5 void QuantumHandler (int *i*)

Implantation On recycle en fin de **readyQ** (p. 48) le thread courant. On choisit le suivant qu'on relance avec un nouveau décomptage de quantum.

```
{
  FredThread old;
  /* interruption de fin de quantum
    on est masqué contre l'it quantum*/
  /* printf("+"); */
  FredMaskOn(); /* début d'exclusion mutuelle */
  current->err=errno; /* sauver errno */
  old=current;
  FredQueuePutLast(&readyQ,current);
  current=FredQueueGetFirst(&readyQ);
  if(current!=old)FredContextSwitch(old->ctx,current->ctx);
  /*
    on reviendra ici lors de son réveil :
    on relance le décomptage du quantum
  */
  FredCountDownStart(quantum);
  errno=current->err;/* restaurer errno */
  FredMaskOff();/* fin d'exclusion mutuelle */
}
```

5.18.2.6 void TimeSharingInit (long int *uS*)

Implantation On initialise le quantum avec une valeur donnée et on attache le traitement d'interruption de fin de quantum au décompteur.

```
{
  /* attaché à une interuption un traitant d'IT */
  FredCountDownAttach(QuantumHandler);
  /* valeur du quantum */
  quantum=uS;
}
```

5.18.2.7 void* watchDogCode (void * *x*)

Il évite d'avoir une file **readyQ** (p. 48) vide à traiter.

Implantation C'est un boucle infinie qui ne fait que passer la main à un autre thread prêt.

```
{
  while(1){
    FredThreadYield();
    /* attraper les interruptions externes */
```

```
  }
  /* on ne revient jamais la */
  FredAbort(" erreur fatale dans chien de garde");
}
```

5.18.2.8 void WatchDogInit ()

Implantation

```
{
  watchdog=FredThreadSpawn(20000,watchDogCode,(void*)0,(void**)0);
}
```

5.18.3 Documentation de la variable

5.18.3.1 FredThread current

5.18.3.2 FredQueue deadQ

5.18.3.3 FredThread leon

Comme un thread terminant ne peut libérer la pile dans laquelle il s'exécute, ce travail est assuré par un thread spécifique de nettoyage : **leon** (p. 48) . Il attend qu'un thread mourant le réveille dans une file d'attente spécifique **leonQ** (p. 44) . Il trouve les threads décédés dans la file **deadQ** (p. 48) (cf **leonCode**() (p. 45)) Il assure aussi la détection de terminaison.

5.18.3.4 int volatile nbFredThreads

Seuls les threads créés par **FredThreadSpawn**() (p. 17) sont comptés. Ce compteur sert à détecter la terminaison du programme FRED (cf **FredWait-Terminate**() (p. 14)).

5.18.3.5 FredQueue readyQ

Elle n'est jamais vide du fait de l'existence du chien de garde **watchdog** (p. 48)

5.18.3.6 FredThread watchdog

Son rôle est d'éviter une file **readyQ** (p. 48) vide. Il ne fait que passer la main. (cf. **watchDogCode**() (p. 47)).

6 Documentation de la classe

6.1 Référence de la structure FredCell

descripteur de cellule de liste

`#include <list.h>`

Attributs Publics

- FredCell * **next**
- FredCell * **pred**

6.1.1 Description détaillée

Dans une liste circulaire à 1 élément, celui ci est son propre successeur et prédécesseur. Il est donc doublement rechainé sur lui-même.

6.1.2 Documentation des données imbriquées

6.1.2.1 struct FredCell* next

élément suivant de file

6.1.2.2 struct FredCell* pred

élément pécédent de file

La documentation associée à cette structure a été générée à partir du fichier suivant :
- **list.h**

6.2 Référence de la structure FredList

descripteur de liste

`#include <list.h>`

Attributs Publics

- **FredCell * last**

6.2.1 Description détaillée

Une liste circulaire pointe sur le dernier élément. Pour une liste vide, ce derrnier élément est nul.

6.2.2 Documentation des données imbriquées

6.2.2.1 FredCell* last

dernier élément de file

La documentation associée à cette structure a été générée à partir du fichier suivant :
- **list.h**

6.3 Référence de la structure FredSemaphore

Descripteur du sémaphore.

`#include <sema.h>`

Attributs Publics

- volatile int **counter**
- **FredQueue queue**

6.3.1 Description détaillée

Un sémaphore est décrit par un compteur et une file d'attente. La valeur initiale du compteur doit être >= 0

Implantation Le compteur représente le nombre de jetons si le compteur est >=0 et le nombre de threads en attente s'il est <0.

6.3.2 Documentation des données imbriquées

6.3.2.1 volatile int counter

compteur de jeton

6.3.2.2 FredQueue queue

queue des threads en attente

La documentation associée à cette structure a été générée à partir du fichier suivant :
- **sema.h**

6.4 Référence de la structure FredThreadDesc

Descripteur de threads.

Attributs Publics

- **FredCELL**

- char * **stack**
- int **stacksz**
- **FredRunnable entry**
- void * **val**
- void ** **resref**
- int **err**
- FredContext **ctx**

6.4.1 Description détaillée

Le champ **FredThreadDesc : :err** (p. 51) est la sauvegarde de la variable globale errno qui contient le code d'erreur du programme. On la sauve et la restaure à chaque commutation de thread.

6.4.2 Documentation des données imbriquées

6.4.2.1 FredContext ctx

contexte registres processeur

6.4.2.2 FredRunnable entry

procédure appelée au lancement du thread

6.4.2.3 int err

sauvegarde code erreur errno

6.4.2.4 FredCELL

pour mettre les thread en file

6.4.2.5 void** resref

adresse resultat

6.4.2.6 char* stack

adr zone pile

6.4.2.7 int stacksz

taille de pile

6.4.2.8 void* val

argument

La documentation associée à cette structure a été générée à partir du fichier suivant :
- **thread.c**

7 Documentation du fichier

7.1 Référence du fichier contextswitch.h

Inclusion et choix de la commutation de contexte.

`#include <setjmp.h>`

Définitions des types

– typedef jmp_buf **LinuxIA32FredContext**
  Contexte de sauvegarde : un jmp_buf de linux.

– typedef jmp_buf **SolarisSparc32FredContext**
  Contexte de sauvegarde : un jmp_buf de solaris.

– typedef int **IA32FredContext** [1]
  Contexte de sauvegarde : un pointeur de sommet de pile où on été empilés tous les autres registes.

Fonctions

– void **LinuxIA32FredContextRestore** (**LinuxIA32FredContext** restore)
  Restaurer un contexte processeur. Le contexte courant est perdu.

– void **LinuxIA32FredContextInit** (**LinuxIA32FredContext** ctx, char *SP, void(*entry)())
  Initialiser un contexte processeur.

– void **SolarisSparc32FredContextRestore** (**SolarisSparc32FredContext** restore)
  Restaurer un contexte processeur. Le contexte courant est perdu.

– void **IA32FredContextRestore** (**IA32FredContext** restore)
  Restaurer un contexte processeur. Le contexte courant est perdu.

– void **IA32FredContextInit** (**IA32FredContext** ctx, char *SP, void(*entry)())
  Initialiser un contexte processeur.

7.1.1 Description détaillée

Ce fichier inclut les définitions des différentes interfaces de commutation de contexte. Il permet de choisir par le biais d'une définition de macro variable l'implantation à utiliser (voir le fichier makefile)

– _SETJMP_LINUX_IA32_ pour une implantation sous Linux/IA32 avec setjmp/longjmp

– _SETJMP_SOLARIS_SPARC32_ pour une implantation sous Solaris/Sparc32 avec setjmp/longjmp
– _ARCH_IA32_ pour une implantation native sous Linux/IA32

7.2 Référence du fichier countdown.h

Inclusion du décompteur.

Fonctions

– long **FredCountDownStart** (long uS)
Initier un décomptage.

– long **FredCountDownStop** ()
Arrêter un décomptage.

– long **FredCountDownGet** ()
Lecture du décompteur.

– void **FredCountDownAttach** (void(*handler)(int))
Attache un traitant de fin de décomptage.

7.2.1 Description détaillée

7.3 Référence du fichier Cstack.h

Inclusion de gestion de pile C.

Définitions des macros

– #define **FredMIN_STACK** 8192
Taille minimum de pile de thread.

– #define **FredMIN_FRAME** 60
Taille minimum de trame de procédure C.

– #define **FredSTACK_SIZE**(sz) (((FredMIN_STACK>(sz)) ?FredMIN_-STACK :(sz))+FredMIN_FRAME)
Calcul d'une taille de pile correcte.

– #define **FredMAKE_SP**(a, sz) ((a)+(sz))
Calcul du pointeur de pile.

7.3.1 Description détaillée

Le fichier **Cstack.h** contient les macros définissant la taille minimum de pile **FredMIN_STACK** (p. 53) , la taille minimum de trame **FredMIN_FRAME**

(p. 53) , le calcul d'une taille de pile viable **FredSTACK_SIZE**() (p. 53) et le calcul d'un pointeur de pile à partir d'une adresse de bloc libre **FredMAKE_-SP**() (p. 53) .

7.4 Référence du fichier fred.h

Fichier racine d'inclusion des types et profils de l'interface FRED.

`#include "thread.h"`

`#include "list.h"`

`#include "mem.h"`

`#include "io.h"`

7.4.1 Description détaillée

7.5 Référence du fichier inspect_stack.c

Programme d'analyse de la pile C.

`#include <malloc.h>`

`#include <stdio.h>`

`#include <signal.h>`

7.5.1 Description détaillée

Ce programme détermine pour la machine qui l'exécute :
- le sens de croissance d'une pile C,
- la taille d'une trame de procédure à 1 paramètre pointeur et un résultat int
- la taille nécessaire au traitement d'un signal (SIGSTKSZ)

Il génère alors le fichier **Cstack.h** qui contient les macros définissant la taille minimum de pile, le calcul d'une taille de pile viable et le calcul d'un pointeur de pile à partir d'une adresse de bloc libre.

```
#include <malloc.h>
#include <stdio.h>
#include <signal.h>
int minFrame; int minStack;
int pushdownstack; /* sens de croissance de la pile :
    1 des adresses hautes vers basses
    0 des adresses  basses vers hautes
*/
/*******************************************************
* Analyse de la pile (sens de croissance .. taille trame min. etc..
********************************************************/
char* mark0; /* première marque */
char * mark1; /* seconde  marque */
void setMark1(int * pt){ int c; mark1=(char*)&c;}
void  setMinFrame(int * pt){ setMark1(pt);}
void setMark0(){ int c; mark0=(char*)&c;setMinFrame(&c);}
```

```
int  main(){
  FILE *f;
  f=fopen("Cstack.h","w");
  fprintf(f,"/* fichier généré par inspect_stack.c : NE PAS TOUCHER */ \n"   );
  /* génération documentation pour doxygen */
  fprintf(f,"/*! \\file Cstack.h \n"   );
  fprintf(f,"\\brief Inclusion de gestion de pile C \n \n"   );
  fprintf(f,"Le fichier Cstack.h  contient les macros   définissant la taille minimum de pile #FredMIN_STACK , la taille minimum de trame #FredMIN_FRAME , le calcul d'une taille de pile viable FredSTACK_SIZE() et le calcul d'un pointeur de pile à partir d'une adresse de bloc libre FredMAKE_SP() .\n");
  fprintf(f,"*/ \n"   );

  /* calcul taille pile pour traitement d'un signal */
  minStack=SIGSTKSZ;
  fprintf(f,"/*! \\brief Taille minimum de pile de thread  \n"   );
  fprintf(f,"*/ \n"   );
  fprintf(f,"#define FredMIN_STACK %d \n",SIGSTKSZ   );
  printf(" taille pile signal min %d \n",SIGSTKSZ );
  /* calcul de la plus petite trame de procédure et sens de croissance*/
  setMark0();
  if( mark0 > mark1 ){
    pushdownstack=1;
    minFrame= mark0-mark1;
    printf(" pile decroissante et trame minimum %d \n", minFrame);
    fprintf(f,"/*! \\brief Taille minimum de trame de procédure C  \n"   );
    fprintf(f,"*/ \n"   );
    fprintf(f,"#define FredMIN_FRAME %d \n", minFrame  );
    fprintf(f,"/*! \\brief Calcul d'une taille de pile correcte  \n"   );
    fprintf(f,"*/ \n"   );
    fprintf(f,"#define FredSTACK_SIZE(sz) (((FredMIN_STACK>(sz))?FredMIN_STACK:(sz))+FredMIN_FRAME)    \n"  );
    fprintf(f,"/*! \\brief Calcul du pointeur de pile  \n"   );
    fprintf(f,"*/ \n "   );
    fprintf(f,"#define FredMAKE_SP(a,sz) ((a)+(sz))  \n"  );
  }else if ( mark0 < mark1 ){
    pushdownstack=0;
    minFrame= mark1-mark0;
    printf(" pile croissante et trame minimum %d \n", minFrame);
    fprintf(f,"/*! \\brief Taille minimum de trame de procédure C  \n"   );
    fprintf(f,"*/ \n"   );
    fprintf(f,"#define FredMIN_FRAME %d \n", minFrame  );
    fprintf(f,"/*! \\brief Calcul d'une taille de pile correcte  \n"   );
    fprintf(f,"*/ \n"   );
    fprintf(f,"#define FredSTACK_SIZE(sz) (((FredMIN_STACK>(sz))?FredMIN_STACK:(sz))+FredMIN_FRAME)    \n"  );
    fprintf(f,"/*! \\brief Calcul du pointeur de pile  \n"   );
    fprintf(f,"*/ \n"   );
    fprintf(f,"#define FredMAKE_SP(a,sz) (a)  \n"  );
  }else printf( " PB inspection de pile ");
    fclose(f);
}
```

7.6 Référence du fichier interrupt.h

Inclusion de la gestion des traitants d'interruption.

Fonctions

– void **FredSignalAttach** (int sig, void(∗handler)(int))

Définir un traitant d'interruption.

– void **FredSignalsInit** ()
Initialise les traitants par défaut.

7.6.1 Description détaillée

7.7 Référence du fichier io.h

Inclusion des E/S.

`#include <stdio.h>`

Fonctions

– int **FredPrintf** (const char ∗format,...)
Imprime **atomiquement** *sur la sortie standard.*

– int **FredFprintf** (FILE ∗stream, const char ∗format,...)
Imprime **atomiquement** *dans un fichier.*

– int **FredSprintf** (char ∗str, const char ∗format,...)
Imprime **atomiquement** *dans un buffer.*

– int **FredScanf** (const char ∗format,...)
lit et décompose **atomiquement** *un texte depuis un fichier source*

– int **FredFscanf** (FILE ∗stream, const char ∗format,...)
lit et décompose **atomiquement** *un texte depuis l'entrée standard*

– int **FredSscanf** (char ∗str, const char ∗format,...)
lit et décompose **atomiquement** *un texte depuis un buffer*

7.7.1 Description détaillée

7.8 Référence du fichier list.h

Inclusion de la gestion de liste.

Composants

– struct **FredCell**
descripteur de cellule de liste

– struct **FredList**
descripteur de liste

Définitions des macros

- #define **NIL** ((**FredCell***)0)
  macro pour définir le pointeur vide
- #define **FredCELL** **FredCell** _cell
  macro pour définir une structure comme une cellule de liste
- #define **FredListEmpty**(Q) (((Q) → last) == (**FredCell***)0)
  Test de file vide.

Définitions des types

- typedef FredCell **FredCell**
  descripteur de cellule de liste
- typedef FredList **FredList**
  descripteur de liste

Fonctions

- void **FredListPutLast** (**FredList** *q, **FredCell** *b)
  Insertion en queue d'une cellule.
- **FredCell** * **FredListGetFirst** (**FredList** *q)
  Retrait de la cellule de tête.
- void **FredListInit** (**FredList** *q)
  Initialisation de la liste.

7.8.1 Description détaillée

7.9 Référence du fichier mask.h

Inclusion du masquage/démasquage d'interruption.

Fonctions

- void **FredMaskOn** ()
  Début d'exclusion mutuelle par masquage des interruptions.
- void **FredMaskOff** ()
  Fin d'exclusion mutuelle par démasquage des interruptions.
- void **FredMasksInit** ()
  Initialisation des masques de validation/inhibition des interruptions.

7.9.1 Description détaillée

7.10 Référence du fichier mem.h

Inclusion de la gestion mémoire.

Définitions des macros

- #define **FredNEW**(T) (T*)FredAtomicAlloc(sizeof(T))
  Alloue **atomiquement** *un bloc pour un type donné.*
- #define **FredNEW_VECTOR**(T, CT) (T*)FredAtomicCalloc(sizeof(T),(CT))
  Alloue **atomiquement** *un vecteur de blocs pour un type donné.*
- #define **FredDELETE**(X) FredAtomicFree((void*)X)
  Libère **atomiquement** *un bloc ou un vecteur de blocs précédemment alloué.*

Fonctions

- void * **FredAtomicAlloc** (int)
  Alloue **atomiquement** *un bloc d'octets.*
- void * **FredAtomicCalloc** (int, int)
  Alloue **atomiquement** *un vecteur de blocs d'octets.*
- void **FredAtomicFree** (void *)
  Libère **atomiquement** *un bloc précédemment alloué.*

7.10.1 Description détaillée

7.11 Référence du fichier philo.c

Le problème des N philosophes.

`#include "fred.h"`

`#include "sema.h"`

7.11.1 Description détaillée

L'évitement de l'interblocage est assuré par :
- la numérotation des fourchettes
- un ordre de prise des fourchettes droite et gauche dans le sens des numéros croissants et non dans le sens horaire ou anti-horaire.

```
#include "fred.h"
#include "sema.h"
#define NBphilo  7
```

```
FredThread philosophe[NBphilo];
FredSemaphore fourchette[NBphilo];
FredSemaphore finRepas;

 void* codephilo(void * x)
{
  int j,k;
  int i = (int)x;
  FredPrintf(" \n philo %d numero %d demarre \n ",  FredThreadCurrent(),i);
  for( j=0;j<100;j++){
    FredPrintf(" \n philo %d pense %d éme fois \n ", i,j);
    for(k=0;k<1000000;k++); /* ça pense ..*/
    FredThreadYield();
    FredPrintf(" \n philo %d veut manger %d éme fois \n ", i,j);
    if(i<(NBphilo-1)){ /* un droitier */
      FredSemaphoreP(&fourchette[i]);FredSemaphoreP(&fourchette[i+1]);
      FredPrintf(" \n philo droitier %d mange %d éme fois \n ", i,j);
      for( k=0;k<1000000;k++); /* ça mange ..*/
      FredThreadYield();
      FredSemaphoreV(&fourchette[i]);FredSemaphoreV(&fourchette[i+1]);
    }else{ /* le gaucher */
      FredSemaphoreP(&fourchette[0]);FredSemaphoreP(&fourchette[NBphilo-1]);
      FredPrintf(" \n philo GAUCHER %d mange %d éme fois \n ", i,j);
      for( k=0;k<1000000;k++);/* ça mange ..*/
      FredThreadYield();
      FredSemaphoreV(&fourchette[0]);FredSemaphoreV(&fourchette[NBphilo-1]);
    }
  }
  FredSemaphoreV(&finRepas);
  FredPrintf(" \n philo  %d ADIEU \n ", i);
  return (void*) 0;
}

int main ( int argc, char** argv)
{
  int i;
  printf(" C'est parti \n");
  FredInitialize();
  for( i=0;i<NBphilo;i++)
  {
    FredSemaphoreInit(&fourchette[i],1);
    FredPrintf("Initialisation fourchette[%d]]  \n ",  i);
  }
  FredSemaphoreInit(&finRepas,0);
  for( i=0;i<NBphilo;i++)
  {
   philosophe[i]= FredThreadSpawn(20000,codephilo,(void*)i,(void**)0);
   FredPrintf("Lancement philosophe[%d]]  \n ",  i);
  }
  FredPrintf("philodsophes lancés !!!!  \n ");
  for(i=0;i<NBphilo;i++)FredSemaphoreP(&finRepas);
  FredPrintf("philosophes repus !!!!  \n ");
  FredWaitTerminate();
  printf(" C'est termine \n");
  return 0;
}
```

7.12 Référence du fichier sema.h

Inclusion des sémaphores.

`#include "fred.h"`

Composants

- struct **FredSemaphore**
  Descripteur du sémaphore.

Définitions des types

- typedef FredSemaphore **FredSemaphore**
  Descripteur du sémaphore.

Fonctions

- void **FredSemaphoreInit** (**FredSemaphore** ∗s, int i)
  Initialise le descripteur de sémaphore.
- void **FredSemaphoreP** (**FredSemaphore** ∗s)
  Prend un jeton.
- void **FredSemaphoreV** (**FredSemaphore** ∗s)
  Depose un jeton.

7.12.1 Description détaillée

7.13 Référence du fichier thread.c

Implantation de la gestion des threads.

`#include "error.h"`

`#include "thread.h"`

`#include "mask.h"`

`#include "interrupt.h"`

`#include "countdown.h"`

`#include "contextswitch.h"`

`#include "Cstack.h"`

`#include "mem.h"`

`#include <errno.h>`

Composants

- struct **FredThreadDesc**
  Descripteur de threads.

Définitions des types

- typedef FredThreadDesc **FredThreadDesc**
  Descripteur de threads.

Fonctions

- void * **leonCode** (void *x)
  Le code du nettoyeur leon.
- void **LeonInit** ()
  Initialisation de leon le nettoyeur.
- void * **watchDogCode** (void *x)
  Le code du chien de garde watchdog.
- void **WatchDogInit** ()
  Initialisation du chien de garde watchdog.
- void **QuantumHandler** (int i)
  Traitement d'interruption de fin de quantum.
- void **TimeSharingInit** (long int uS)
  Initialisation du temps partagé.
- void **FredThreadSuspend** (**FredQueue** *q)
  Suspendre le thread courant dans une file.
- void **FredThreadWakeUp** (**FredQueue** *q)
  Réveille un thread bloqué.
- void **FredThreadHarness** ()
  Code initial des threads.
- **FredThread FredThreadSpawn** (int stacksz, **FredRunnable** function, void *arg, void **resref)
  Création et lancement d'un thread.
- void **FredThreadExit** (void *res)
  Terminaison immédiate du thread courant.
- void **FredThreadYield** ()
  Le thread courant passe la main à un autre thread.
- **FredThread FredThreadCurrent** ()
  Rend l'identité du thread courant.
- void **MainThreadInit** ()
  Initialisation du pseudo thread principal.

– void **FredInitialize** ()
  Initialisation de FRED.

– void **FredWaitTerminate** ()
  Attente de fin d'exécution de tous les threads créés.

– void **FredAbort** (char ∗mess)
  Arrêt brutal du programme et impression d'un message d'erreur.

Variables

– int volatile **nbFredThreads**
  Compteur du nombre de threads créés non terminés.

– **FredThread current**
  Le thread actif.

– **FredQueue readyQ**
  File d'attente des threads prêts.

– **FredQueue deadQ**
  File d'attente des threads morts dont la pile est à libérer (réservée à **leon** (p. 48) *le nettoyeur) Un thread terminant s'inscrit dans cette file et réveille* **leon** (p. 48) *s'il n'est pas déja réveillé (cf* **leonCode**() (p. 45) *).*

– **FredThread leon**
  Le nettoyeur leon.

– **FredQueue leonQ**
  File d'attente de **leon** (p. 48).

– **FredQueue endQ**
  File d'attente de terminaison (réservée au **mainThread** (p. 62) *).*

– **FredThread mainThread**
  Le thread associé à la procédure **main**().

– **FredThreadDesc mainThreadBlock**
  Descripeur du thread associé à la procédure **main**().

– **FredThread watchdog**
  Thread chien de garde.

– long int **quantum**
  Valeur du quantum en microseconde.

7.13.1 Description détaillée

7.14 Référence du fichier thread.h

Inclusion de la gestion des threads.

```
#include "contextswitch.h"
#include "list.h"
```

Définitions des macros

- #define **FredTHREAD_NIL** (**FredThread**)NIL
  Définition du thread inexistant.
- #define **FredQueueGetFirst**(Q) (**FredThread**)FredListGetFirst((**FredList**∗)Q)
  Retirer le premier thread d'une file d'attente.
- #define **FredQueuePutLast**(Q, th) FredListPutLast((**FredList**∗)Q,(**FredCell**∗)th)
  Insérer un thread déja bloqué dans une file d'attente.
- #define **FredQueueEmpty**(Q) FredListEmpty((**FredList**∗)Q)
  Tester une queue à vide de threads.
- #define **FredQueueInit**(Q) FredListInit((**FredList**∗)Q) ;
  Initialiser une queue vide de threads.
- #define **FredMutexOn**() FredMaskOn()
  Début de section critique.
- #define **FredMutexOff**() FredMaskOff()
  Fin de section critique.

Définitions des types

- typedef void ∗(∗ **FredRunnable**)(void ∗)
  Le type d'une procédure initiale de thread.
- typedef **FredThreadDesc** ∗ **FredThread**
  L'identité d'un thread est un pointeur sur son descripteur.
- typedef **FredList** **FredQueue**
  Définition du type queue de threads.

Fonctions

- void **FredThreadSuspend** (**FredQueue** ∗q)
  Suspendre le thread courant dans une file.
- void **FredThreadWakeUp** (**FredQueue** ∗q)
  Réveille un thread bloqué.
- void **FredThreadExit** (void ∗)
  Terminaison immédiate du thread courant.
- void **FredThreadYield** ()

Le thread courant passe la main à un autre thread.

- void **FredInitialize** ()
  Initialisation de FRED.
- void **FredWaitTerminate** ()
  Attente de fin d'exécution de tous les threads créés.
- void **FredAbort** (char ∗)
  Arrêt brutal du programme et impression d'un message d'erreur.

7.14.1 Description détaillée

Index