Manuel de référence

Généré par Doxygen 1.2.17

Thu Jan 15 15 :49 :02 2004

Table des matières

1 PHIL : Noyau temps réel simple

Auteur :
J. Briat

Date :
Janvier 2004

PHIL est un gros TP d'implantation d'une version simplifié du standard Posix à partir de la base FRED. 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 en temps partagé avec **priorité** ou au gré des synchronisations entre ces threads.

PHIL offre les fonctionnalités primitives d'un noyau temps réel réaliste où les threads sont gérés en temps partagé avec **priorité**. Ses fonctions plus élaborées que celles de FRED sont des extensions des mécanismes de base de FRED. Ces fonctions sont présentées en plusieurs sections :
- **Gestion des threads** (p. 2) : comment créer des threads
- **Exclusion mutuelle** (p. 2) : comment faire une section critique
- **File ou condition d'attente** (p. 3) : comment synchroniser des threads
- **Initialisation & terminaison** (p. 3) : comment définir un programme concurrent PHIL
- **Exemples d'utilisation** (p. 4) : comment utiliser PHIL

1.0.1 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 **PhilRunnable** (p. 11). 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. La création d'un thread nécessite de donner de :
- une procédure de type FredRunnable
- un mode **détaché** ou **joignable**
- une priorité
- une taille de pile
- un paramètre de type (void*)

La procédure **PhilThreadCreate**() (p. 12) crée et lance donc un thread ((c.à.d. le met dans la file des threads prêts à s'exécuter). Le type d'un thread influe sur le critère de terminaison du programme concurrent (**Initialisation & terminaison** (p. 3)).

Un thread se termine par terminaison normale de sa procédure initiale ou par suicide. La procédure **PhilThreadExit**() (p. 12) permet à un thread de se terminer en rendant un résultat de type (void*).

Un thread peut attendre la fin d'un thread **joignable** et récupérer son résultat par **PhilThreadJoin**() (p. 12).

Un thread peut connaitre son identité (reférence sur son descripteur (type **PhilThread** (p. 11))) par la fonction **PhilThreadCurrent**() (p. 12).

Le thread courant peut spontanément abandonner le processeur au profit d'un autre thread prêt au sens des règles d'ordonnancement par priorité par **PhilThreadYield**() (p. 12).

Voir également :
Interface de gestion de threads

1.0.2 Exclusion mutuelle

PHIL offre un mécanisme d'exclusion mutuelle de type verrou (**PhilMutex** (p. 27)). L'usage des mécanismes matériels (inhibition des interruption par exemple) est ainsi réduit à des durées très courtes (implantation des verrous) pour éviter de retarder ou perdre des signaux externes ou bloquer les autres processeurs dans le cas de multiprocesseur.

L'exclusion mutuelle repose sur les 2 procédures **PhilMutexLock**() (p. 14) et **PhilMutexUnlock**() (p. 15). Le verrou ne peut être fermé que par un seul thread qui en devient alors propriétaire. Un verrou ne peut être déverrouillé que par son propriétaire. Il est toujours possible d'imbriquer les sections critiques contrôlées par un même verrou.

```
/* code correct */
....PhilMutexLock(&V1);....; PhilMutexLock(&V1);....PhilMutexUnlock(&V1);....PhilMutexUnlock(&V1);....
```

Un verrou s'initialise **PhilMutexInit**() (p. 14) et se libère **PhilMutexDestroy**() (p. 13) signalant à cette occasion s'il y des threads en attente sur le verrou à ce moment. Les threads en attente sont réveillés et prévenus de la raison de ce réveil.

Voir également :
Interface d'exclusion mutuelle

1.0.3 File ou condition d'attente

Les relations de dépendance et de synchronisation entre threads nécessite de bloquer et réveiller les threads au gré des circonstances. Phil offre le type **PhilCond** (p. 26) qui permet de définir une file d'attente de threads gérée selon la statégie FIFO à priorité. On initialise une condition par **PhilCondInit**() (p. 16).

Le thread courant peut se bloquer sur une condition **PhilCondWait**() (p. 17) jusqu'à ce qu'un thread le réveille. Un thread peut réveiller un thread bloqué sur une condition par **PhilCondSignal**() (p. 17). Le thread réveillé est le + prioritaire. On peut aussi réveiller tous les threads bloqués par **PhilCondBroadcast**() (p. 16). Ces procédures doivent toujours être utilisées en exclusion mutuelle.

Le thread courant peut se bloquer sur une condition **PhilCondTimedWait**() (p. 17) jusqu'à ce qu'un thread le réveille ou qu'une durée limite soit écoulée.

Une condition doit s'initialiser par **PhilCondInit**() (p. 16) et se libérer par **PhilCondDestroy**() (p. 16) signalant à cette occasion s'il y des threads en attente sur la condition.

Voir également :
Interface de gestion des conditions d'attente

1.0.4 Initialisation & terminaison

Pour utiliser PHIL, il faut inclure le fichier **phil.h** pour disposer de toutes les déclaration de types et profils de procédures. La procédure **main**() du programme doit initialiser les variables statiques de PHIL. C'est le rôle de **PhilInitialize**() (p. 10). Puis le programmeur peut initialiser ses propres variables globales et lancer ses threads initiaux, se synchroniser etc.... **PhilTerminate**() (p. 11) lui permet d'attendre la terminaison des threads de calcul (et des threads de calcul créés par ceux-ci et etc....). Sinon, la procédure **main**() termine et tout le programme concurrent aussi. La décision de terminaison ignore les threads **détachés** qui sont alors détruit.

```
int main( int argc, char** argv)
{/*
    PHIL n'est pas utilisable.
    Seul du code C séquentiel est exécutable
 */
  <....>
PhilInitialize();
 /*
    PHIL est partiellement utilisable.
    On peut créer des threads, intialiser des objets satiques,
```

```
    se synchroniser ....
 */
<....>
PhilTerminate();
 /*
    Attente de la terminaison de tous les threads.
    A la reprise, PHIL n'est plus utilisable.
    Seul du code C séquentiel est exécutable.
 */
 <....>
}
```

N'importe quel thread peut terminer brutalement le programme concurrent par **PhilAbort**() (p. 10) en passant en paramètre une chaine de caractère précisant la raison de l'arrêt brutal.

Voir également :

Interface d'initialisation & terminaison

1.1 Exemples d'utilisation

L'exclusion mutuelle par verrou et les conditions sont suffisantes pour reconstruire des opérateurs de synchronisation classique comme le sémaphore de Dijkstra. On peut ensuite traiter un problème classique de synchronisation comme les N philosophes.

1.1.1 Un exemple d'utilisation : le sémaphore

Un sémaphore est un simple sac de jetons que l'on prend ou remet 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 thread le + prioritaire en attente de jeton s'il en existe. Le sémaphore doit être déclaré de type FredSemaphore, alloué en variable globale ou sur le tas. Dans tousles cas, il doit être initialisé à une valeur >=0 via **PhilSemaphoreInit**() (p. 19) et détruit via **PhilSemaphoreDestroy**() (p. 19).

Il peut ensuite être utilisé pour prendre **PhilSemaphoreP**() (p. 19) ou déposer un jeton **PhilSemaphoreV**() (p. 20)

L'implantation nécessite un compteur et une queue de threads en attente (cf. **PhilSemaphore** (p. 28)). 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 atomique et se font donc verrouillés. (cf. **Interface sémaphore** (p. 18)). La suspension se fait via une condition.

1.1.2 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)
{
  int i;
  PhilInitialize();
  for( i=0;i<NBphilo;i++)
    PhilSemaphoreInit(&fourchette[i],1);
  for( i=0;i<NBphilo;i++)
   philosophe[i]= PhilThreadCreate(0, 0, ,20000,codephilo,(void*)i);
  for(i=0;i<NBphilo;i++)PhilThreadJoin(philosophe[i], 0);
  PhilWaitTerminate();
  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,k;
  int i = (int)x;
  for( j=0;j<100;j++){
     /* ça pense ..*/
    if(i<(NBphilo-1)){ /* un droitier */
      PhilSemaphoreP(&fourchette[i]);PhilSemaphoreP(&fourchette[i+1]);
       /* ça mange ..*/
      PhilSemaphoreV(&fourchette[i]);PhilSemaphoreV(&fourchette[i+1]);
    }else{ /* le gaucher */
      PhilSemaphoreP(&fourchette[0]);PhilSemaphoreP(&fourchette[NBphilo-1]);
      /* ça mange ..*/
      PhilSemaphoreV(&fourchette[0]);PhilSemaphoreV(&fourchette[NBphilo-1]);
    }
  }
  return (void*) 0;
}
```

1.2 Implantation de PHIL

L'implantation de PHIL introduit quelques difficultés par rapport à celle de FRED.

- La première porte sur la gestion des priorités. Cela se traduit par l'utilisatioin d'une gestion de liste triée par priorité **Gestion de listes triées** (p. 6) .
- La seconde porte sur une gestion plus fine du temps. On doit en effet gérer le temps partagé, la date (cf **PhilNow**() (p. 18)) et des attentes de durée bornée (cf **PhilCondTimedWait**() (p. 17)). Ceci implique la gestion d'un échéancier **Gestion d'un échéancier** (p. 6) .

– Le dernier problème porte sur l'évitement du phénomène dit **inversion de priorité**. La solution à implanter est celle du **plafonnement dynamique**.

L'implantation de PHIL s'appuie sur des modules intermédiaires ou de modification des modules de FRED.

1.2.1 Gestion de listes triées

Il suffit de modifier la gestion de liste de FRED pour en faire une gestion de liste triée par priorité. On retire toujours en tête mais on trie à l'insertion c'est à dire qu'on trie par priorité et qu'on insère un thread en queue de la sous-liste de priorité égale à sa priorité. **Interface de gestion de listes** (p. 21) La priorité est un entier >=0. La priorité maximum est 0.

1.2.2 Gestion d'un échéancier

La gestion d'un échéancier se ramène à une gestion de date et une gestion d'échéances. Une échéance est une action (**PhilEventHandler** (p. 24)) à effectuer à une date donnée. Ainsi, la fin d'un quantum est une échéance tout comme le réveil d'un thread sur une condition. Le paramètre de l'action est le descripteur (**PhilEvent** (p. 26)) de l'échéance qui se termine.

L'implantation de l'échéancier (**Interface d'un échéancier** (p. 23)) s'appuiera sur la définition du décompteur que l'on trouvera dans FRED. La gestion de la date implique une utilisation permanente du décompteur.

La gestion de la date est relativement simple : la date courante est la date de chargement du décompteur + temps écoulé depuis cette date c'est à dire la valeur de chargement du compteur - le temps résiduel de décompage. A chaque interruption de décomptage, on relance un décomptage. 2 variables d'état suffisent à mémoriser la date de lancement du décomptage et la valeur de décomptage.

Si seule la date était à gérer, il suffirait de charger le décompteur avec la valeur maximum. Ce n'est plus le cas lorsqu'il existe des échéances. Le décompteur doit être chargé avec la valeur exacte qui correspond à l'échéance la plus proche dans le futur de façon à pouvoir exécuter l'action associée à l'échéance. A l'interruption de décomptage correspondant à cette échéance, on doit exécuter l'action et relancer le décomptage s'il y des échéances postérieures. L'action peut aussi avoir rajouter de nouvelles échéances.

Un échéancier va donc se présenter comme une liste triée par date croissante de cellules décrivant une échéance ou plutot une liste triée par date croissante de sous listes d'échéances tombant à la même date.

Le décompteur sera toujours relancé à partir de la tête de liste. Sur une interruption de fin de décomptage, toutes les actions échues seront exécutées.

Des échéances peuvent être annulées avant la date prévue. Il faut les retirer de l'échéancier. On doit éventuellement relancer le décomptage si cette échéance était la + proche.

1.2.3 Gestion par priorité des threads

La gestion par priorité des threads implique de maintenir la propriété que le thread actif est le + prioritaire des threads prêts. Ceci implique que lors de tout réveil de thread, on évalue la priorité du thread courant contre la priorité du thread réveillé pour savoir si le processeur doit être donné au réveillé plutot que conservé par le thread actif. Lorsqu'un verrou est relaché, ce verrou est donné au + prioritaire des demandeurs qui peut être aussi + prioritaire que le libérateur du verrou. Cette situation implique aussi une commutation de thread.

La complication principale provient du phénomène **d'inversion de priorité**. Il est demandé d'implanter une solution par **plafonnement dynamique**. Le principe est de détecter une situation de conflit entre un thread propriétaire du verrou et un thread demandant le verrou. Ce conflit peut se produire lors d'un **PhilMutexLock**() (p. 14) fait par le thread + prioritaire ou lors d'un **PhilCondSignal**() (p. 17) ou **PhilCondBroadcast**() (p. 16) fait par le thread - prioritaire. Dans ce cas, il réveille un(tous les) thread(s) qui redemandent le verrou. La solution est de hausser la priorité du thread propriétaire au niveau de celle du + prioritaire pour accélérer sa sortie de l'exclusion mutuelle.

Lorsque le conflit est détecté par le thread le plus prioritaire, celui-ci se bloque après avoir hausser la priorité du propriétaire. Si celui-ci est dans la file des prêts, il peut tourner s'il est remonté en tête des prêts. Ce n'est pas toujours le cas car il peut être aussi en attente sur un autre verrou ou une condition. La aussi, il doit être reclassé dans la liste d'attente. Il est donc **nécessaire** de connaitre la file dans laquelle se trouve un thread (chainage inverse).

L'accroisement de priorité n'est pas la seule difficulté. C'est aussi le cas du rétablissement de la priorité antérieure. En effet, du fait de l'inclusion possible des sections critiques et du caractère dynamique des conflits d'inversion, il est nécessaire de recalculer la priorité d'un thread à la libération d'un verrou. Pour ce faire, il faut gérer par thread la **liste des verrous pris** par ce thread. Sa priorité doit être la priorité maximum des threads en attente sur ces verrous. A la libération d'un verrou, on doit parcourir la liste des verrous pris par le thread pour calculer la priorité maximum des threads en attente.

1.3 Organisation du travail

Le travail comprend plusieurs actions permettant de faire une démonstration de l'état de la réalisation. Voici un exemple de découpage :

- **action1.1** : Modifier FRED pour implanter l'API PHIL (**Interface de gestion de threads** (p. 11)). La statégie de gestion du processeur de FRED est inchangée. Il s'agit d'implanter le verrou récursif, la condition (sans le **PhilCondTimedWait**() (p. 17)), la création et la synchronisation de terminaison des threads.
- **action1.2** : Modification de la gestion de liste FRED pour faire la gestion de liste triée au sens de PHIL (**Interface de gestion de listes** (p. 21))
- **action2.1** <== Fusion action1.1 et action1.2. On obtient une gestion de thread par priorité sans traitement de l'inversion de priorité et sans le **PhilCondTimedWait**() (p. 17)

- **action2.2** Réalisation de la gestion d'échéancier (**Interface d'un échéancier** (p. 23)) à partir du décompteur de FRED et de la gestion de liste triée de PHIL
- **action3.1** <== Fusion action2.1 et action2.2 En traitant la fin de quantum comme une fin d'échéance et le réveil d'une condition sur fin d'échéance, on obtient un noyau PHIL sans traitement de l'inversion de priorité.
- **action3.2** Extension de action2.1 pour traiter l'inversion de priorité.
- **action4** <== Fusion action3.1 et 3.2 Le noyau PHIL complet

Il y a une activité parallèle et continue de développement des tests destinés à vérifier la correction de la réalisation. Indépendemment des tests unitaires de fonctions, on réalisera un test de l'API PHIL via l'implantation d'exemples **classiques** :
- producteur consommateur
- lecteur rédacteur
- N philosophes (directement avec verrous et conditions)

Les tests d'échéancier et de priorités sont laissés à la discrétion des implémenteurs.

Avertissement :

La fusion est une opération délicate qui demande à avoir été préparée!!!

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 PHIL

Ce module définit l'interface applicative de PHIL.

Modules

- **Interface d'initialisation & terminaison**
  Ce module définit comment initier et terminer un programme concurrent PHIL.
- **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 créer des verrous, les prendre ou les libérer.

- **Interface de gestion des conditions d'attente**
  Ce module définit comment se bloquer ou réveiller dans une file d'attente.

5.1.1 Description détaillée

5.2 Interface d'initialisation & terminaison

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

Fonctions

- int **PhilInitialize** ()
  Initialisation de PHIL.

- int **PhilTerminate** ()
  Terminaison de PHIL.

- void **PhilAbort** (char *mess)
  Terminaison forçée de PHIL.

5.2.1 Description détaillée

Pour savoir comment l'utiliser, voir **Initialisation & terminaison** (p. 3)

5.2.2 Documentation de la fonction

5.2.2.1 void PhilAbort (char * *mess*)

Paramètres :
  mess le message d'erreur

Un thread aborte le programme PHIL.

5.2.2.2 int PhilInitialize ()

Renvoie :
  un code d'erreur :
  - 0 OK
  - >0 erreur à l'initialisation

5.2.2.3 int PhilTerminate ()

Renvoie :
un code d'erreur :
- 0 OK
- >0 erreur à la terminaison

Le thread principal attend la terminaison de tous les threads joignables. Les threads détachés sont ignorés.

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 types

- typedef void *(* **PhilRunnable**)(void *)
  Le type d'une procédure exécutable par un thread.
- typedef PhilThreadDesc * **PhilThread**
  L'identité d'un thread est un pointeur sur son descripteur.

Fonctions

- **PhilThread PhilThreadCreate** (int type, int priority, int stacksize, **PhilRunnable** start_routine, void *arg)
  Création d'un thread.
- int **PhilThreadJoin** (**PhilThread** th, void **return_val)
  Joindre un thread.
- void **PhilThreadExit** (void *return_val)
  Terminer le thread courant.
- **PhilThread PhilThreadCurrent** ()
  Obtenir l'identité du thread courant.
- void **PhilThreadYield** ()
  Passer la main à un autre thread.

5.3.1 Description détaillée

Pour savoir comment l'utiliser, voir **Un exemple d'utilisation : le sémaphore** (p. 4) **Les N philosophes** (p. 4)

5.3.2 Documentation de la fonction

5.3.2.1 PhilThread PhilThreadCreate (int *type*, int *priority*, int *stacksize*, PhilRunnable *start_routine*, void * *arg*)

Paramètres :

type le thread est joignable 0 ou détaché 1

priority >=0 définit la priorité du thread créé. 0 est la priorité maximum.

stacksize taille de pile demandée

start_routine procédure initiale du thread

arg paramètre initial

Renvoie :

l'identité du thread ou 0 si echec de création

Un thread détaché ne peut être joint. Il est ignoré lors de la détection de terminaison qui ne prend en compte que les threads joignables.

5.3.2.2 PhilThread PhilThreadCurrent ()

Renvoie :

identité du thread courant

5.3.2.3 void PhilThreadExit (void * *return_val*)

Paramètres :

return_val adresse de la valeur résultat.

5.3.2.4 int PhilThreadJoin (PhilThread *th*, void ** *return_val*)

Paramètres :

th le thread joint

return_val où mettre l'adresse du résultat du thread joint. Si 0, le résultat est ignoré.

Renvoie :

un code d'erreur :
- 0 OK
- 1 le thread joint est inexistant

Un thread détaché ne peut être join. Il est ignoré lors de la détection de terminaison qui ne prend en compte que les threads joignables.

5.3.2.5 void PhilThreadYield ()

Le passage de main ne peut se faire qu'au sein de threads de même priorité ou de priorités supérieures au thread courant.

5.4 Interface d'exclusion mutuelle

Ce module définit comment créer des verrous, les prendre ou les libérer.

Composants

- struct **PhilMutex**
  Descripteur du verrou.

Définitions des types

- typedef PhilMutex **PhilMutex**
  Descripteur du verrou.

Fonctions

- int **PhilMutexInit** (**PhilMutex** ∗verrou)
  Initialiser un verrou.
- int **PhilMutexDestroy** (**PhilMutex** ∗verrou)
  Destruction du verrou.
- int **PhilMutexLock** (**PhilMutex** ∗verrou)
  Prendre le verrou.
- int **PhilMutexTryLock** (**PhilMutex** ∗verrou)
  Essai de fermeture du verrou.
- int **PhilMutexUnlock** (**PhilMutex** ∗verrou)
  Ouverture du verrou.

5.4.1 Description détaillée

Pour savoir comment l'utiliser, voir **Un exemple d'utilisation : le sémaphore** (p. 4) **Les N philosophes** (p. 4)

5.4.2 Documentation du type

5.4.2.1 typedef struct PhilMutex PhilMutex

L'identité d'un verrou est un pointeur sur son descripteur

5.4.3 Documentation de la fonction

5.4.3.1 int PhilMutexDestroy (PhilMutex ∗ *verrou*)

Paramètres :
verrou adresse du verrou

Renvoie :
un code d'erreur :
- 0 OK
- 1 le verrou est inexistant ou non initialisé
- 2 Le thread n'est pas le propriétaire du verrou
- 3 des threads attendent le verrou.

On ne peut détruire qu'un verrou dont on est propriétaire. S'il y a des threads en attente, ils sont réveillés et la destruction leur est signalée.

5.4.3.2 int PhilMutexInit (PhilMutex * *verrou*)

Paramètres :
verrou adresse du verrou

Renvoie :
0 OK 1 verrou inexistant (verrou==0)

5.4.3.3 int PhilMutexLock (PhilMutex * *verrou*)

Paramètres :
verrou adresse du verrou

Renvoie :
un code d'erreur :
- 0 OK
- 1 le verrou est inexistant ou non initialisé
- 2 le verrou a été détruit (**PhilMutexDestroy**() (p. 13)) durant l'attente

Le verrou peut se verrouiller de façon récursive.Il est alloué au thread + prioritaire qui devient propriétaire.

5.4.3.4 int PhilMutexTryLock (PhilMutex * *verrou*)

Paramètres :
verrou adresse du verrou

Renvoie :
un code d'erreur :
- 0 OK verrou fermé
- 1 le verrou est inexistant ou non initialisé
- 2 echec de fermeture

Le verrou peut se verrouiller de façon récursive.Il est alloué au thread + prioritaire qui devient propriétaire. Il n'y a jamais blocage.

5.4.3.5 int PhilMutexUnlock (PhilMutex * *verrou*)

Paramètres :
verrou adresse du verrou

Renvoie :
un code d'erreur :
- 0 OK
- 1 le verrou est inexistant ou non initialisé
- 2 Le thread n'est pas le propriétaire du verrou

Comme le verrou peut être verrouillé de façon récursive, il nécessite le même nombre de déverrouillages que de verrouillages pour être ouvert. Par ailleurs, on ne peut déverrouiller qu'un verrou dont on est propriétaire.

5.5 Interface de gestion des conditions d'attente

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

Composants

- struct **PhilCond**
  Descripteur de condition.

Définitions des types

- typedef PhilCond **PhilCond**
  Descripteur de condition.

Fonctions

- int **PhilCondInit** (**PhilCond** *cond)
  Initialiser une condition.
- int **PhilCondDestroy** (**PhilCond** *cond)
  Destruction de la condition.
- int **PhilCondSignal** (**PhilCond** *cond)
  Signaler une condition.
- int **PhilCondBroadcast** (**PhilCond** *cond)
  Signaler une condition.
- int **PhilCondWait** (**PhilCond** *cond, **PhilMutex** *mutex)
  Attendre une condition.
- int **PhilCondTimedWait** (**PhilCond** *cond, **PhilMutex** *mutex, long int us)
  Attendre une condition pour une durée bornée.

– long int **PhilNow** ()
Gestion de l'heure.

5.5.1 Description détaillée

Pour savoir comment l'utiliser, voir **Un exemple d'utilisation : le sémaphore** (p. 4) **Les N philosophes** (p. 4)

5.5.2 Documentation du type

5.5.2.1 typedef struct PhilCond PhilCond

L'identité d'une condition est un pointeur sur son descripteur

5.5.3 Documentation de la fonction

5.5.3.1 int PhilCondBroadcast (PhilCond ∗ *cond*)

Paramètres :
cond adresse de condition

Renvoie :
un code d'erreur :
– 0 OK
– 1 condition inexistante ou non initialisée

Tous les threads en attente sont réveillés.

5.5.3.2 int PhilCondDestroy (PhilCond ∗ *cond*)

Paramètres :
cond adresse de la condition

Renvoie :
un code d'erreur :
– 0 OK
– 1 la condition est inexistante ou non initialisée
– 2 des threads attendaient sur la condition au moment de la destruction

S'il y a des threads en attente, ils sont réveillés et la destruction leur est signalée.

5.5.3.3 int PhilCondInit (PhilCond ∗ *cond*)

Paramètres :
cond adresse du condition

Renvoie :
0 OK 1 condition inexistante

5.5.3.4 int PhilCondSignal (PhilCond * *cond*)

Paramètres :

cond adresse de condition

Renvoie :

un code d'erreur :
- 0 OK
- 1 condition inexistante ou non initialisée

Le thread en attente le + prioritaire est réveillé.

5.5.3.5 int PhilCondTimedWait (PhilCond * *cond*, PhilMutex * *mutex*, long int *us*)

Paramètres :

cond adresse du condition

mutex verrou à libérer au blocage et à reprendre au réveil

us durée maximum d'attente en microsecondes

Renvoie :

un code d'erreur :
- 0 OK
- 1 la condition est inexistante ou non initialisée
- 2 la condition est détruite (**PhilCondDestroy**() (p. 16)) durant l'attente
- 3 durée limite atteinte

On peut sortir de l'attente à la suite d'un **PhilCondSignal**() (p. 17), un **PhilCondBroadcast**() (p. 16), d'une destruction de la condition **PhilCondDestroy**() (p. 16) ou de l'épuisement du délai.

5.5.3.6 int PhilCondWait (PhilCond * *cond*, PhilMutex * *mutex*)

Paramètres :

cond adresse de condition

mutex verrou à libérer au blocage et à reprendre au réveil

Renvoie :

un code d'erreur :
- 0 OK
- 1 la condition est inexistante ou non initialisée
- 2 la condition est détruite (**PhilCondDestroy**() (p. 16)) durant l'attente

On peut sortir de l'attente à la suite d'un **PhilCondSignal**() (p. 17), un **PhilCondBroadcast**() (p. 16) ou une destruction de la condition **PhilCondDestroy**() (p. 16).

5.5.3.7 long int PhilNow ()

Renvoie :
la date en microseconde

L'origine des temps sera le démarrage du programme PHIL

5.6 Interface sémaphore

Une implémentation classique du classique sémaphore de Dijkstra.

Composants

- struct **PhilSemaphore**
  Descripteur du sémaphore.

Définitions des types

- typedef PhilSemaphore **PhilSemaphore**
  Descripteur du sémaphore.

Fonctions

- int **PhilSemaphoreInit** (**PhilSemaphore** ∗s, int i)
  Initialise le descripteur de sémaphore.
- int **PhilSemaphoreDestroy** (**PhilSemaphore** ∗s)
  Détruit le sémaphore.
- int **PhilSemaphoreP** (**PhilSemaphore** ∗s)
  Prend un jeton.
- int **PhilSemaphoreV** (**PhilSemaphore** ∗s)
  Depose un jeton.

5.6.1 Description détaillée

5.6.2 Documentation du type

5.6.2.1 typedef struct PhilSemaphore PhilSemaphore

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.

5.6.3 Documentation de la fonction

5.6.3.1 int PhilSemaphoreDestroy (PhilSemaphore * *s*)

Paramètres :

s adresse descripteur sémaphore

Renvoie :

code d'erreur 0 OK 1 erreur

Avertissement :

la valeur initiale du sémaphore doit être positive ou nulle

```
{
  /* le semaphore existe et la valeur initiale est >=0 */
  if (s !=(PhilSemaphore *) 0){
    if(PhilMutexDestroy(&(s->lock)))return 1 ;
    if(PhilCondDestroy(&(s->queue)))return 1 ;
    return 0;
  }
  return 1;
}
```

5.6.3.2 int PhilSemaphoreInit (PhilSemaphore * *s*, int *i*)

Paramètres :

s adresse descripteur sémaphore

i nombre initial de jetons

Renvoie :

code d'erreur 0 OK 1 erreur

Avertissement :

la valeur initiale du sémaphore doit être positive ou nulle

```
{
  /* le semaphore existe et la valeur initiale est >=0 */
  if ((s !=(PhilSemaphore *) 0)&&(i>=0)){
    s->counter = i;
    if(PhilMutexInit(&(s->lock)))return 1 ;
    if(PhilCondInit(&(s->queue)))return 1 ;
    return 0;
  }
  return 1;
}
```

5.6.3.3 int PhilSemaphoreP (PhilSemaphore * *s*)

Paramètres :

s adresse descripteur sémaphore

Renvoie :
code d'erreur 0 OK 1 erreur S'il n'y a pas de jeton, le thread se bloque. Il sera réveillé à son tour (file d'attente FIFO à priorité) quand un jeton lui sera donné.

Implantation

```
{
  if(!s)return 1;
  if(PhilMutexLock(&(s->lock)))return 1;/* début d'exclusion mutuelle */
  s->counter--;
  while(s->counter < 0){
    /* pas de jeton :  on suspend le thread actif   */
    if(PhilCondWait(&(s->queue),&(s->lock))){
      PhilMutexUnlock(&(s->lock));
      return 1  ;
    }
  }
  if(PhilMutexUnlock(&(s->lock)))return 1;/* fin d'exclusion mutuelle */
  return 0;
}
```

5.6.3.4 int PhilSemaphoreV (PhilSemaphore * *s*)

Paramètres :
s adresse descripteur sémaphore

Renvoie :
code d'erreur 0 OK 1 erreur S'il y a des threads en attente d'un jeton, on réveille le + prioritaire.

Implantation

```
{
  if(!s)return 1;
  if(PhilMutexLock(&(s->lock)))return 1;/* début d'exclusion mutuelle */
  s->counter++;
  if(s->counter <=0){
    /* il y a des demandeurs en attente */
    if(PhilCondSignal(&(s->queue)))return 1;
  }
  if(PhilMutexUnlock(&(s->lock)))return 1;/* fin d'exclusion mutuelle */
  return 0;
}
```

5.7 Implémentation de PHIL

Ce module regroupe les sous modules nécessaire à la mise en oeuvre de PHIL.

Modules

- **Interface de gestion de listes**
- **Interface d'un échéancier**

5.7.1 Description détaillée

5.8 Interface de gestion de listes

Composants

- struct **PhilCell**
  descripteur de cellule de liste
- struct **PhilList**
  descripteur de liste

Définitions des macros

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

Définitions des types

- typedef PhilCell **PhilCell**
  descripteur de cellule de liste
- typedef PhilList **PhilList**
  descripteur de liste

Fonctions

- void **PhilListPutLast** (**PhilList** *q, **PhilCell** *b)
  Insertion en queue d'une cellule.
- **PhilCell** * **PhilListGetFirst** (**PhilList** *q)
  Retrait de la cellule de tête.
- void **PhilListInit** (**PhilList** *q)
  Initialisation de la liste.

5.8.1 Description détaillée

C'est une gestion de liste circulaire doublement chaînée triée par priorité croissante. La priorité est un entier >=0. La priorité maximum est 0.

5.8.2 Documentation de la macro

5.8.2.1 #define NIL ((philCell*)0)

5.8.2.2 #define PhilCELL PhilCell _cell

Pour définir qu'une structure quelconque peyt être insérer dans une liste, il suffit d'insérer PhilCELL en premiere définition des composants de la structure

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

5.8.2.3 #define PhilListEmpty(Q) (((Q) → last) == (PhilCell*)0)

5.8.3 Documentation du type

5.8.3.1 typedef struct PhilCell PhilCell

La priorité est un entier >=0. La priorité maximum est 0. Dans une liste circulaire à 1 élément, celui ci est son propre successeur et prédécesseur. l est donc doublement rechainé sur lui-même.

5.8.3.2 typedef struct PhilList PhilList

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

5.8.4 Documentation de la fonction

5.8.4.1 PhilCell* PhilListGetFirst (PhilList * *q*)

Paramètres :
 q liste où retirer

Renvoie :
 cellule retirer

On retire toujours en tête où se trouve la cellule la + prioritaire.

5.8.4.2 void PhilListInit (PhilList * *q*)

Paramètres :
 q liste à initialiser

5.8.4.3 void PhilListPutLast (PhilList * *q*, PhilCell * *b*)

L'insertion en queue place la nouvelle cellule à la suite des cellules de même priorité (ou de priorité supérieure) et devant les cellules de priorité inférieure.

5.9 Interface d'un échéancier

Composants

- struct **PhilEvent**
  Descripteur de cellule de ech On réutilise la cellule de liste triée par priorité. La priorité est la date. On rajoute le traitement de fin d'échéance.

Définitions des macros

- #define **PhilEVENT PhilEvent** _event
  macro pour définir une structure comme une échéance

Définitions des types

- typedef void(* **PhilEventHandler**)(struct **PhilEvent** *)
  Prototype d'un traitant d'échéance.
- typedef PhilEvent **PhilEvent**
  Descripteur de cellule de ech On réutilise la cellule de liste triée par priorité. La priorité est la date. On rajoute le traitement de fin d'échéance.
- typedef **PhilList PhilEventList**
  descripteur d'échéancier

Fonctions

- void **PhilEventListInsert** (**PhilEventList** *q, **PhilEvent** *b)
  Insertion d'un évenement.
- void **PhilEventListPurge** (**PhilEventList** *q, **PhilEventList** *past)
  Obtenir les échéances échues.
- void **PhilEventListRetract** (**PhilEventList** *q, **PhilEvent** *b)
  Retrait d'une échéance quelconque.
- long int **PhilEventListNow** ()
  Date courante.
- void **PhilEventListInit** (**PhilEventList** *q, long int date)
  Initialisation de la liste.

5.9.1 Documentation de la macro

5.9.1.1 #define PhilEVENT PhilEvent _event

Pour définir qu'une structure quelconque peut être une échéance, il suffit d'insérer PhilEVENT en premiere définition des composants de la structure

```
struct MyEvent{
  PhilEVENT;
  int x;
  int y;
  ...
};
```

5.9.2 Documentation du type

5.9.2.1 typedef struct PhilEvent PhilEvent

5.9.2.2 typedef void(* PhilEventHandler)(struct PhilEvent *)

L'action appelée en fin d'échéance reçoit l'échéance en paramètre.

5.9.2.3 typedef PhilList PhilEventList

C'est une liste

5.9.3 Documentation de la fonction

5.9.3.1 void PhilEventListInit (PhilEventList * *q*, long int *date*)

Paramètres :

q liste à initialiser

date date initiale en microsecondes

initialise l'échéance et lance le calcul de la date

5.9.3.2 void PhilEventListInsert (PhilEventList * *q*, PhilEvent * *b*)

Paramètres :

q échéancier où insérer

b échéance à insérer

L'insertion place la nouvelle échéance à la suite des échéances de même date (ou de date inférieue) et devant celles de dates supérieures. On doit éventuellement (si insertion en tête) relancer le décomptage avec une nouvelle valeur.

5.9.3.3 long int PhilEventListNow ()

Renvoie :
la date en microseconde

5.9.3.4 void PhilEventListPurge (PhilEventList * *q*, PhilEventList * *past*)

Paramètres :
- ***q*** échéancier à consulter
- ***past*** liste des échéances échues (vide à l'appel)

On obtient la liste des échéances <= à la date courante. Cette liste peut être vide. Cette fonction sert à vider l'échéancier après une interruption de fin d'échéance. On relance éventuellement le décompteur.

5.9.3.5 void PhilEventListRetract (PhilEventList * *q*, PhilEvent * *b*)

Paramètres :
- ***q*** liste où retirer
- ***b*** cellule à retirer

Une échéance est annulée (par ex. un signal est arrivé avant la fin du délai d'une attente sur une condition). On doit la retirer de la liste. Si cette échéance était la plus proche, on peut recalculer éventuellement la prochaine échéance et relancer le décompte.

6 Documentation de la classe

6.1 Référence de la structure PhilCell

descripteur de cellule de liste

`#include <priolist.h>`

Attributs Publics

- PhilCell * **next**
- PhilCell * **pred**
- long int **prio**

6.1.1 Description détaillée

La priorité est un entier >=0. La priorité maximum est 0. Dans une liste circulaire à 1 élément, celui ci est son propre successeur et prédécesseur. l est donc doublement rechainé sur lui-même.

6.1.2 Documentation des données imbriquées

6.1.2.1 struct PhilCell* next

élément suivant de file

6.1.2.2 struct PhilCell* pred

élément pécédent de file

6.1.2.3 long int prio

priorité de l'élément

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

6.2 Référence de la structure PhilCond

Descripteur de condition.

`#include <phil.h>`

6.2.1 Description détaillée

L'identité d'une condition est un pointeur sur son descripteur

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

6.3 Référence de la structure PhilEvent

Descripteur de cellule de ech On réutilise la cellule de liste triée par priorité. La priorité est la date. On rajoute le traitement de fin d'échéance.

`#include <ech.h>`

Attributs Publics

- **PhilCELL**
- **PhilEventHandler action**

6.3.1 Description détaillée

6.3.2 Documentation des données imbriquées

6.3.2.1 PhilEventHandler action

action à échéance

6.3.2.2 PhilCELL

chainage de liste et date

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

6.4 Référence de la structure PhilList

descripteur de liste

`#include <priolist.h>`

Attributs Publics

- **PhilCell * last**

6.4.1 Description détaillée

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

6.4.2 Documentation des données imbriquées

6.4.2.1 PhilCell* last

dernier élément de file

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

6.5 Référence de la structure PhilMutex

Descripteur du verrou.

`#include <phil.h>`

6.5.1 Description détaillée

L'identité d'un verrou est un pointeur sur son descripteur

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

6.6 Référence de la structure PhilSemaphore

Descripteur du sémaphore.

`#include <psema.h>`

Attributs Publics

- volatile int **counter**
- **PhilMutex lock**
- **PhilCond queue**

6.6.1 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.

6.6.2 Documentation des données imbriquées

6.6.2.1 volatile int counter

compteur de jeton

6.6.2.2 PhilMutex lock

verrou d'exclusion mutuelle

6.6.2.3 PhilCond queue

queue des threads en attente

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

7 Documentation du fichier

7.1 Référence du fichier ech.h

Gestion de la date et d'un échéancier.

Composants

- struct **PhilEvent**
  Descripteur de cellule de ech On réutilise la cellule de liste triée par priorité. La priorité est la date. On rajoute le traitement de fin d'échéance.

Définitions des macros

– #define **PhilEVENT PhilEvent** _event
macro pour définir une structure comme une échéance

Définitions des types

– typedef void(∗ **PhilEventHandler**)(struct **PhilEvent** ∗)
Prototype d'un traitant d'échéance.

– typedef PhilEvent **PhilEvent**
Descripteur de cellule de ech On réutilise la cellule de liste triée par priorité. La priorité est la date. On rajoute le traitement de fin d'échéance.

– typedef **PhilList PhilEventList**
descripteur d'échéancier

Fonctions

– void **PhilEventListInsert** (**PhilEventList** ∗q, **PhilEvent** ∗b)
Insertion d'un évenement.

– void **PhilEventListPurge** (**PhilEventList** ∗q, **PhilEventList** ∗past)
Obtenir les échéances échues.

– void **PhilEventListRetract** (**PhilEventList** ∗q, **PhilEvent** ∗b)
Retrait d'une échéance quelconque.

– long int **PhilEventListNow** ()
Date courante.

– void **PhilEventListInit** (**PhilEventList** ∗q, long int date)
Initialisation de la liste.

7.1.1 Description détaillée

7.2 Référence du fichier phil.h

Inclusion de l'interface PHIL.

Composants

– struct **PhilCond**
Descripteur de condition.

– struct **PhilMutex**
Descripteur du verrou.

Définitions des types

- typedef void *(* **PhilRunnable**)(void *)
  Le type d'une procédure exécutable par un thread.
- typedef PhilThreadDesc * **PhilThread**
  L'identité d'un thread est un pointeur sur son descripteur.
- typedef PhilMutex **PhilMutex**
  Descripteur du verrou.
- typedef PhilCond **PhilCond**
  Descripteur de condition.

Fonctions

- **PhilThread PhilThreadCreate** (int type, int priority, int stacksize, **PhilRunnable** start_routine, void *arg)
  Création d'un thread.
- int **PhilThreadJoin** (**PhilThread** th, void **return_val)
  Joindre un thread.
- void **PhilThreadExit** (void *return_val)
  Terminer le thread courant.
- **PhilThread PhilThreadCurrent** ()
  Obtenir l'identité du thread courant.
- void **PhilThreadYield** ()
  Passer la main à un autre thread.
- int **PhilMutexInit** (**PhilMutex** *verrou)
  Initialiser un verrou.
- int **PhilMutexDestroy** (**PhilMutex** *verrou)
  Destruction du verrou.
- int **PhilMutexLock** (**PhilMutex** *verrou)
  Prendre le verrou.
- int **PhilMutexTryLock** (**PhilMutex** *verrou)
  Essai de fermeture du verrou.
- int **PhilMutexUnlock** (**PhilMutex** *verrou)
  Ouverture du verrou.
- int **PhilCondInit** (**PhilCond** *cond)
  Initialiser une condition.
- int **PhilCondDestroy** (**PhilCond** *cond)
  Destruction de la condition.

- int **PhilCondSignal** (**PhilCond** ∗cond)
  Signaler une condition.
- int **PhilCondBroadcast** (**PhilCond** ∗cond)
  Signaler une condition.
- int **PhilCondWait** (**PhilCond** ∗cond, **PhilMutex** ∗mutex)
  Attendre une condition.
- int **PhilCondTimedWait** (**PhilCond** ∗cond, **PhilMutex** ∗mutex, long int us)
  Attendre une condition pour une durée bornée.
- long int **PhilNow** ()
  Gestion de l'heure.
- int **PhilInitialize** ()
  Initialisation de PHIL.
- int **PhilTerminate** ()
  Terminaison de PHIL.
- void **PhilAbort** (char ∗mess)
  Terminaison forçée de PHIL.

7.2.1 Description détaillée

7.3 Référence du fichier philo.c

Le problème des N philosophes.

`#include "phil.h"`

`#include "psema.h"`

7.3.1 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 "phil.h"
#include "psema.h"


#define NBphilo  7
PhilThread philosophe[NBphilo];
PhilSemaphore fourchette[NBphilo];
```

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

```
int main ( int argc, char** argv)
{
  int i;
  printf(" C'est parti \n");
  PhilInitialize();
  for( i=0;i<NBphilo;i++)
  {
    PhilSemaphoreInit(&fourchette[i],1);
    PhilPrintf("Initialisation fourchette[%d]]  \n ",  i);
  }
  for( i=0;i<NBphilo;i++)
  {
   philosophe[i]= PhilThreadCreate(0, 0,20000,codephilo,(void*)i);
   PhilPrintf("Lancement philosophe[%d]]  \n ",  i);
  }
  PhilPrintf("philodsophes lancés !!!!  \n ");
  for(i=0;i<NBphilo;i++)PhilThreadJoin(philosophe[i], 0);
  PhilPrintf("philosophes repus !!!!  \n ");
  PhilWaitTerminate();
  printf(" C'est termine \n");
  return 0;
}
```

7.4 Référence du fichier priolist.h

inclusion de la gestion par priorité de liste

Composants

- struct **PhilCell**
  descripteur de cellule de liste
- struct **PhilList**
  descripteur de liste

Définitions des macros

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

Définitions des types

- typedef PhilCell **PhilCell**
  descripteur de cellule de liste
- typedef PhilList **PhilList**
  descripteur de liste

Fonctions

- void **PhilListPutLast** (**PhilList** ∗q, **PhilCell** ∗b)
  Insertion en queue d'une cellule.
- **PhilCell** ∗ **PhilListGetFirst** (**PhilList** ∗q)
  Retrait de la cellule de tête.
- void **PhilListInit** (**PhilList** ∗q)
  Initialisation de la liste.

7.4.1 Description détaillée

7.5 Référence du fichier psema.h

Inclusion des sémaphores.

```
#include "phil.h"
```

Composants

– struct **PhilSemaphore**
  Descripteur du sémaphore.

Définitions des types

– typedef PhilSemaphore **PhilSemaphore**
  Descripteur du sémaphore.

Fonctions

– int **PhilSemaphoreInit** (**PhilSemaphore** ∗s, int i)
  Initialise le descripteur de sémaphore.

– int **PhilSemaphoreDestroy** (**PhilSemaphore** ∗s)
  Détruit le sémaphore.

– int **PhilSemaphoreP** (**PhilSemaphore** ∗s)
  Prend un jeton.

– int **PhilSemaphoreV** (**PhilSemaphore** ∗s)
  Depose un jeton.

7.5.1 Description détaillée

Index