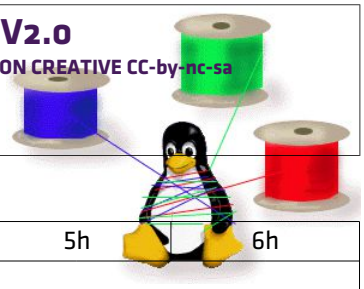
Responsable pédagogique	**AF**	AM	OP	PM
Période	Sem1	Sem2	**Sem3**	Sem4
	Cours/TD		TP	
Volume horaire	**6**		**8 (Philosophes)**	

TD Thread V2.0



Indicateur temporel (hors rédaction du compte-rendu) :

questions	1h	2h	3h	4h	5h	6h
Thread						
La protection de ressources						
Le rendez-vous						
Producteurs/**Consommateurs**						

Documents à rendre : qualité et pertinence du compte-rendu (analyse, commentaires, problèmes rencontrés,...).

Critères d'évaluation : Respect du Cahier des Charges et respect du plan du TP, qualité des logiciels écrit.

Acquis de ce TD : Programmation système par thread, sémaphores, algorithme producteurs/consommateurs

Présentation

L'objectif de ce TD est d'apprendre à utiliser les threads ainsi que tous les mécanismes permettant la programmation concurrente.

Le travail à faire est encadré. Le reste est à lire attentivement.

Avant Propos : Généralités

Les processus légers (en anglais, thread), également appelés fils d'exécution, sont similaires aux processus en cela qu'ils représentent tous deux l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur ces exécutions semblent se dérouler en parallèle. Toutefois là où chaque processus possède sa propre mémoire virtuelle, les processus légers appartenant au même processus père partagent une même partie de sa mémoire virtuelle.

Comme utilisation typique de processus légers on peut citer une interface graphique d'un programme où les interactions de l'utilisateur avec le processus, par l'intermédiaire des périphériques d'entrée, sont gérées par un processus léger, tandis que les calculs lourds (en terme de temps de calcul) sont gérés par un ou plusieurs autres processus légers. Cette technique de conception de logiciel est avantageuse dans ce cas, car l'utilisateur peut continuer d'interagir avec le programme même lorsque celui-ci sera en train d'exécuter une tâche. Une application pratique se retrouve dans les traitements de texte où la correction orthographique est exécutée tout en permettant à l'utilisateur de continuer à entrer son texte.

L'utilisation des processus légers permet donc de rendre l'utilisation d'une application plus fluide, car il n'y a plus de blocage durant les phases de traitements intenses.

Les processus légers se distinguent du multitâche plus classique par le fait que deux processus sont typiquement indépendants et peuvent interagir uniquement à travers une API fournie par le système telle que IPC. D'un autre côté les processus légers partagent une information sur l'état du processus, des zones de mémoires ainsi que d'autres ressources. Sur certains systèmes la commutation de contexte (context switch) entre deux processus légers est moins coûteuse que la commutation de contexte entre deux processus. On peut y voir un avantage de la programmation utilisant des threads multiples ou bien une faiblesse des dits systèmes d'exploitation concernant la commutation de contexte entre deux processus.

Dans certains cas les programmes utilisant des processus légers sont plus rapides que des

programmes architecturés plus classiquement, en particulier sur les machines comportant plusieurs processeurs. Hormis le problème du coût de la commutation de contexte, qui dépend en grande partie du système d'exploitation utilisé, le principal surcoût à l'utilisation de processus multiples provient de la communication entre processus séparés. En effet le partage de certaines ressources entre processus légers permet une communication plus efficace entre les différents threads d'un processus. Là où deux processus séparés doivent utiliser un mécanisme fourni par le système pour communiquer, les processus légers partagent tout ou partie de l'état du processus.

D'un autre côté la programmation utilisant des processus légers est plus difficile, l'accès à certaines ressources partagées doit être restreint par le programme lui-même, pour éviter que tout ou partie de l'état d'un processus ne devienne temporairement incohérent, tandis qu'un autre processus léger va avoir besoin de consulter cette portion de l'état du processus. Il est donc obligatoire de mettre en place des mécanismes de synchronisation (à l'aide de sémaphores par exemple). La complexité des programmes utilisant des processus légers est aussi nettement plus grande que celle des programmes déférant le travail à faire à plusieurs processus plus simples. Cette complexité accrue, lorsqu'elle est mal gérée lors de la phase de conception ou d'implémentation d'un programme, peut conduire à de multiples problèmes.

remarques : Il ne faut pas confondre la technologie Hyperthreading incluse dans certains processeurs Intel avec les processus légers. Cette technologie permet en effet aussi bien l'exécution simultanée de processus distincts que de processus légers. Toute machine comportant des processeurs multiples (SMP) ou des processeurs intégrant l'HyperThreading permettra l'exécution plus rapide de programmes utilisant des processus légers aussi bien que des processus multiples.

Threads

Création de threads

Les threads se créée avec pthread_create :

```
(extrait du man)
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void*),
                   void *restrict arg);
```

Les arguments :

- l'argument thread est l'adresse d'un pthread_t ou pthread_create vas stocker le numéro de thread créé.
- attr est pointeur sur pthread_attr_t qui permet de définir le comportement du thread. Si on y place NULL, les attributs par défauts seront utilisés.
- start_routine est un pointeur de fonction de type void *(*)(void*). C'est à dire que c'est l'adresse d'une fonction qui reçois un pointeur non typé et renvoie également un pointeur non typé.
- arg est l'argument transmis à la fonction start_routine.

La fonction pthread_create renvoie 0 si il n'y a pas eu d'erreur.

Une fois l'exécution de cette méthode terminé, la fonction passé en paramètre est exécuté en parallèle du fils d'exécution appelant.

remarque : Si notre fonction ne reçois pas de paramètres et/ou ne retourne rien, il est possible de transtyper (cast).

Synchronisation des threads

Il existe plusieurs manière de synchroniser des threads. Nous allons ici nous intéresser à l'attente de la fin d'exécution d'un thread par la fonction pthread_join.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

Cette fonction attend la fin d'exécution du thread et stocke la variable de retour dans value_ptr. Si nous ne sommes pas intéressé par le retour de la fonction exécuter par la thread nous pouvons mettre NULL.

Programme de base

soit le programme suivant :

```
// deux thread, sync on end ---- /!\ lier avec -lpthread
#include <pthread.h>
#include <stdio.h>
int* f1(int* pi){
        static int res = 0;
        res = *pi+1 ;
        printf("th1 : res = %3d, pi = %p, &res = %p, *pi = %3d\n",
               res, pi, &res, *pi);
        return &res ;
}
int* f2(int* pi){
        static int res = 0;
        res = *pi+1 ;
        printf("th2 : res = %3d, pi = %p, &res = %p, *pi = %3d\n",
           res,pi,&res,*pi);
        return &res;
}
int main(){
        int a, b ;
        int *ret1 = 0, *ret2 = 0 ;
        pthread_t th1, th2 ;

        a = 10 ;
        b = 20 ;

        /* création des deux threads */


        /* Attendre la fin de leur exécutions */


        printf("main) th1 : *ret1 = %3d, &res = %p\n",*ret1,ret1);
        printf("main) th2 : *ret2 = %3d, &res = %p\n",*ret2,ret2);

        return 0 ;
}
```

Étudier le programme présenté et faire en sorte que :

- deux threads soit créées, une exécutant f1, l'autre exécutant f2,
- f1 recevra l'adresse de a et f2 celle de b,
- f1 renverra la résultat sur ret1 et f2 sur ret2.
- Repérer les différentes adresses, dans votre compte-rendu vous établirez un dessin des variables utilisées.

(ceci suppose d'attendre la fin des threads dans le main)

Les sémaphores / verrous

Introduction

Un sémaphore est une variable protégée (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente. Le sémaphore a été inventé par Edsger Dijkstra et utilisé pour la première fois dans le système d'exploitation THEOS.

Les sémaphores fournissent la solution la plus courante pour le fameux problème du « dîner des philosophes » que vous aurez la possibilité d'étudier dans les chapitres suivants, bien qu'ils ne permettent pas d'éviter tous les inter-blocages (ou deadlocks).

Opérations

Les trois opérations prises en charge sont Init, P et V. P et V signifient en néerlandais **Proberen**, tester, et **Verhogen**, incrémenter (en français "Puis-je?" et "Vas-y!"). La valeur d'un sémaphore est le nombre d'unités de ressource (exemple : imprimantes...) libres; s'il n'y a qu'une ressource, un sémaphore à système numérique binaire avec les valeurs 0 ou 1 est utilisé.

- L'opération **P** est en attente jusqu'à ce qu'une ressource soit disponible, ressource qui sera immédiatement allouée au processus courant.
- **V** est l'opération inverse; elle rend simplement une ressource disponible à nouveau après que le processus a terminé de l'utiliser.
- **Init** est seulement utilisé pour initialiser le sémaphore. Cette opération ne doit être utilisée qu'une seule et unique fois.

Les opérations P et V doivent être indivisibles (atomique), ce qui signifie que les différentes opérations ne peuvent pas être exécutées plusieurs fois de manière concurrente. Un processus qui désire exécuter une opération qui est déjà en cours d'exécution par un autre processus doit attendre que le premier termine.

Dans les livres en anglais, les opérations V et P sont quelques fois appelées respectivement up et down. En conception logicielle, elles sont appelées signal et wait ou release et take.

Pour éviter l'attente, un sémaphore peut avoir une file de processus associée (généralement une file du type FIFO). Si un processus exécute l'opération P sur un sémaphore qui a la valeur zéro, le processus est ajouté à la file du sémaphore. Quand un autre processus incrémente le sémaphore en exécutant l'opération V, et qu'il y a des processus dans la file, l'un d'eux est retiré de la file et reprend la suite de son exécution.

Ceci étant dit, passons aux choses sérieuses.

Utilisations

Les sémaphores sont utilisés dans beaucoup d'algorithmes de programmation concurrente : protection de ressources, rendez vous, producteurs/consommateurs, maîtres/esclaves, pipe-line, philosophes, ...

Nous allons en étudier trois différents.

Les sémaphores POSIX

Création/destruction du sémaphore

```
int sem_init(sem_t *sem, int pshared, unsigned int valeur);
```

sem_init initialise le sémaphore pointé par sem. Le compteur associé au sémaphore est initialisé à valeur. L'argument pshared indique si le sémaphore est local au processus courrant (pshared est zéro) ou partagée entre plusieurs processus (pshared n'est pas nulle). LinuxThreads ne gère actuellement pas les sémaphores partagés entre plusieurs processus, donc sem_init renvoie toujours l'erreur ENOSYS si pshared n'est pas nulle.

```
int sem_destroy(sem_t *sem);
```

destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Exemple :

```
sem_t sem_name ;
sem_init(&sem_name, 0, 10);
/.../
sem_destroy(&sem_name);
```

P – prise du sémaphore

```
int sem_wait(sem_t * sem);
```

Exemple : `sem_wait(&sem_name);`

```
int sem_trywait(sem_t * sem);
```

sem_wait suspend le thread appelant jusqu'à ce que le sémaphore pointé par sem aît un compteur non nul. Alors, le compteur du sémaphore est atomiquement décrémenté.

sem_trywait est une variante non bloquante de sem_wait. Si le sémaphore pointé par sem a une valeur non nulle, le compteur est atomiquement décrémenté et sem_trywait retourne immédiatement 0. Si le compteur du sémaphore est zéro, sem_trywait retourne immédiatement en indiquant l'erreur EAGAIN.

V – Rendre le sémaphore

```
int sem_post(sem_t * sem);
```

sem_post incrémente atomiquement le compteur du sémaphore pointé par sem. Cetts fonction ne bloque jamais et peut être utilisée de manière fiable dans un gestionnaire de signaux.

Exemple : `sem_post(&sem_name);`

les Mutex POSIX

Création/destruction du mutex

```
int pthread_mutex_init(  pthread_mutex_t *restrict mutex,
                const pthread_mutexattr_t *restrict attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Exemple :

```
   pthread_mutex_t lock;
   if (pthread_mutex_init(&lock, NULL) != 0)
        {
         perror("\n mutex init failed\n");
         return 1;
    }
   /.../
   pthread_mutex_destroy(&lock);
```

P – prise du mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Exemple :

```
pthread_mutex_lock(&lock);
```

V – Rendre le mutex

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exemple :

```
pthread_mutex_unlock(&lock);
```

La protection de ressources

Soit le code suivant :

```
#include ...
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define TABSZ 4

...
...


void f1(int* tab){
    int i ;
    printf("\t\t\tf1 started...\n");
    for( i = 0 ; i<10 ; i++) {

        /* P */
        printf("\t\t\t\t\tf1 tentative de prise du mutex \n");
      ...
        printf("\t\t\t\t\tf1 prise effective du mutex (P)\n\n\n\n\n");


        tab[0] ++ ;
        tab[1] = tab[0] + 1 ;
        sleep(rand()%4); // changement de contexte artificiel
        tab[2] = tab[1] + 1 ;
        tab[3] = tab[2] + 1 ;

        /* V */
      ...
        printf("\t\t\t\t\tf1 rendre mutex (V) \n");

    }
}

void f2(int* tab){
    int i ;
    printf("\t\t\t\t\tf2 started...\n");
    for( i = 0 ; i<10 ; i++) {

        /* P */
        printf("\t\t\t\t\t\t\t\t\t\t\tf2 tentative de prise du mutex \n");
      ...
        printf("\t\t\t\t\t\t\t\t\t\t\tf2 prise effective du mutex (P)\n\n\n\n\n");

        tab[0] = tab[0] + ((i%2)?100:-100) ;
        tab[1] = tab[1] + ((i%2)?100:-100) ;
        sleep(rand()%4); // changement de contexte artificiel
        tab[2] = tab[2] + ((i%2)?100:-100) ;
        tab[3] = tab[3] + ((i%2)?100:-100) ;

        /* V */
      ...
        printf("\t\t\t\t\t\t\t\t\t\t\tf2 rendre mutex (V) \n");

    }
}
```

```
void printTab(int* tab){
    int i ;

    /* P */
    printf("PrintTab tentative de prise du mutex \n");
  ...
    printf("PrintTab prise effective du mutex (P)\n");

    for (i=0;i<TABSZ;i++)
        printf("[%d] = %d\n",i,tab[i]) ;
    printf("\n") ;

    /* V */
  ...
    printf("PrintTab rendre mutex (V) \n");

}
int main(){
    int tab[TABSZ]={0,0,0,0} ; // ressource partagée
    int i = 0;
    int err ;

    /* création des deux threads et du mutex/sémaphore*/

  ...

  ...

  ...


    for(i = 0; i < 10 ; i++){
        printTab(tab);
        sleep(1);
    }
    /* Attendre la fin de leur exécutions + destruction de ce qui doit
     l'être*/

    ...
    ...

    printTab(tab);

    ...

    return 0 ;
}
```

Compléter le code présenté pour

- Créer deux thread pour f1 et f2, exécuter le programme (vous commenterez les « ... » manquant) et noter le résultat, vous semble-t-il cohérent ?
- Créer et initialiser le sémaphore *lock.* Protéger l'accès au tableau dans les trois fonctions.

  Le résultat final doit être :
  [0] = 10
  [1] = 11
  [2] = 12
  [3] = 13

Le rendez-vous

Le rendez vous est en faite une barrière de synchronisation.

Pour réaliser une barrière de synchronisation, il faut disposer de deux sémaphores et d'une variable :

- Un sémaphore **MUTEX** (initialisé à 1) protégeant la variable.
- Un sémaphore **ATTENTE** (initialisé à 0) permettant de mettre en attente les tâches.
- Une variable **Nb_Att** (initialisée à 0) permettant de compte le nombre de tâches déjà arrivées à la barrière de synchronisation.

Il faut encore définir la constante **N** qui indique le nombre de tâches devant arriver à la barrière avant de l'ouvrir.

Barrière :

```
P(MUTEX)
Nb_Att++
SI Nb_Att==N ALORS
   POUR I DE 1 à N-1 FAIRE
      V(ATTENTE)
   FIN POUR
   Nb_Att=0
   V(MUTEX)
SINON
   V(MUTEX)
   P(ATTENTE)
FIN SI
```

- Créer un programme démonstrateur du rendez vous sur la base du programme suivant.
- Ce programme contiendra un main, une fonction barrière ainsi que f1, f2 et f3.

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

pthread_t tid[3];

...

...

void f1(){
       sleep(1); printf("(1)f1 synchronizing ...\n");
       sleep(1); printf("(2)f1 synchronizing ...\n");
       sleep(1); printf("(3)f1 synchronizing ...\n");

       /* Attente du RDV */
       printf("f1 synchronization done ...\n");
       /* instructions ... */
}
void f2(){
       sleep(1); printf("(1)f2 synchronizing ...\n");
       sleep(1); printf("(2)f2 synchronizing ...\n");
       sleep(1); printf("(3)f2 synchronizing ...\n");
       sleep(1); printf("(5)f2 synchronizing ...\n");
       sleep(1); printf("(6)f2 synchronizing ...\n");

       /* Attente du RDV */
       printf("f2 synchronization done ...\n");
       /* instructions ... */
}
void f3(){
       sleep(1); printf("(1)f3 synchronizing ...\n");
```

```
        /* Attente du RDV */
        printf("f3 synchronization done ...\n");
        /* instructions ... */
}


int main(){
    int i = 0;
    int err ;

...

    err = pthread_create(&(tid[0]), NULL, (void *(*)(void *))f1, NULL);
    if (err != 0)
        printf("\ncan't create thread :[%s]", strerror(err));

    err = pthread_create(&(tid[1]), NULL, (void *(*)(void *))f2, NULL);
    if (err != 0)
        printf("\ncan't create thread :[%s]", strerror(err));

    err = pthread_create(&(tid[2]), NULL, (void *(*)(void *))f3, NULL);
    if (err != 0)
        printf("\ncan't create thread :[%s]", strerror(err));

    /* Attendre la fin de leur exécutions + destruction de ce qui doit
     l'être*/

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_join(tid[2], NULL);


...


    return 0 ;
}
```

Modèle Producteurs / Consommateurs (rédacteurs /lecteurs)

Le problème

Supposons qu'une base de données ait des lecteurs et des rédacteurs, et qu'il faille programmer les lecteurs et les rédacteurs de cette base de données.

Les contraintes sont les suivantes :

- plusieurs lecteurs doivent pouvoir lire la base de données en même temps ;
- si un rédacteur est en train de modifier la base de données, aucun autre utilisateur (ni rédacteur, ni même lecteur) ne doit pouvoir y accéder.

Solutions

Il est assez simple de faire en sorte que le rédacteur soit mis en attente tant qu'il y a encore des lecteurs.

Mais cette solution présente de gros problèmes, si le flux de lecteurs est régulier : le rédacteur pourrait avoir à patienter un temps infini.

Il existe donc une deuxième solution, qui consiste à mettre en attente tous les lecteurs ayant adressé leur demande d'accès après celle d'un rédacteur.

Edsger Dijkstra, qui a formulé ce problème, propose de le résoudre au moyen de sémaphores.

Solution avec utilisation des sémaphores et priorité aux lecteurs

La solution suivante permet de résoudre le problème des lecteurs et des rédacteurs en donnant priorité aux lecteurs. Cette solution nécessite trois sémaphores et une variable, à savoir :

- Un sémaphore M_Lect, initialisé à 1 qui permet de protéger la variable Lect. Il s'agit donc d'un mutex.
- Un sémaphore M_Red, initialisé à 1 qui permet de bloquer les tâches de rédaction. Il s'agit donc à

nouveau d'un mutex.

- Un sémaphore Red, initialisé à 1 qui permet aussi de bloquer les tâches de rédaction.
- Une variable Lect qui compte le nombre de lecteurs.

Cette solution utilise les quatre méthodes suivantes ...

Commencer une lecture

```
Commencer_Lire :
   P(M_Lect)
   Lect++
   SI Lect==1 ALORS
      P(Red)
   FIN SI
   V(M_Lect)
```

Cette méthode incrémente le nombre de lecteurs ; Ensuite, s'il s'agit du premier lecteur à essayer d'entrée, elle n'autorise l'entrée que s'il n'y a pas de rédaction en cours. Dans le cas contraire, la méthode doit attendre que la rédaction soit finie. L'utilisation de deux sémaphores pour les rédacteurs permet d'induire la priorité aux lecteurs.

Finir une lecture

```
Finir_Lire :
   P(M_Lect)
   Lect--
   SI Lect==0 ALORS
      V(Red)
   FIN SI
   V(M_lect)
```

Cette méthode décrémente le nombre de lecteurs. Ensuite, s'il s'agit du dernier lecteur à sortir, elle autorise les rédacteurs à entrer (si nécessaire).

Commencer une écriture

```
Commencer_Ecrire
   P(M_Red)
   P(Red)
```

Finir une écriture

```
Finir_Ecrire
   V(Red)
   V(M_Red)
```

Implémentation

Votre travail consiste à créer 5 threads. Deux types de threads seront présents : les lecteurs (3) et les producteurs (2).

Les producteurs produiront chacun un entier (aléatoire ou incrémenté) et les rédacteurs copieront ces deux entiers en local pour les afficher.

La structure partagée sera une variable globale de type ***int tab[2]***, l'indice 0 étant réservé à un des rédacteurs et l'indice 1 à l'autre.

Algorithme des producteurs

```
Commencer_Ecrire()
Écrire entier dans la zone partagée
Finir_Ecrire()
```

Algorithme rédacteurs

```
Commencer_Lire()
Copier les deux entiers partagé en local
Finir_Lire()
```

Philosophes – Thread/Mutex et C++

Pour se faire plaisir...

Le problème

La situation est la suivante :

- cinq philosophes (initialement mais il peut y en avoir beaucoup plus) se trouvent autour d'une table ;
- chacun des philosophes a devant lui un plat de riz ;
- à gauche de chaque assiette se trouve une baguette.

Un philosophe n'a que trois états possibles :

- penser pendant un temps indéterminé ;
- être affamé (pendant un temps déterminé et fini sinon il y a famine) ;
- manger pendant un temps déterminé et fini.

Des contraintes extérieures s'imposent à cette situation :

- quand un philosophe a faim, il va se mettre dans l'état « affamé » (hungry) et attendre que les baguettes soient libres ;
- pour manger, un philosophe a besoin de deux baguettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à gauche de celle de son voisin de droite (c'est-à-dire les deux baguettes qui entourent sa propre assiette);
- si un philosophe n'arrive pas à s'emparer d'une baguette, il reste affamé pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour.

Remarques

- Les philosophes, s'ils agissent tous de façons naïves et identiques, risquent fort de se retrouver en situation d'interblocage. En effet, il suffit que chacun saisisse sa baguette de gauche et, que ensuite, chacun attende que sa baguette de droite se libère pour qu'aucun d'entre eux ne puisse manger, et ce pour l'éternité.
- On considère qu'un philosophe qui meurt (crash du processus) reste dans une phase « penser » infiniment. Il en résulte donc un problème : quid d'un philosophe qui meurt avec ses baguettes en main ?
- Ce problème beaucoup, plus complexe qu'il n'en a l'air, est l'un des plus intéressants parmi les problèmes de systèmes distribués.

blague : Socrate raconte une bonne blague, se sert une rasade de ciguë et meurt avec sa baguette gauche en main, empêchant définitivement Voltaire de manger

Solutions

- L'une des principales solutions à ce problème est celle du sémaphore, proposée également par Dijkstra.
- Une autre solution consiste à attribuer à chaque philosophe un temps de réflexion aléatoire en

cas d'échec.

- Il existe des compromis qui permettent de limiter le nombre de philosophes embêtés par une telle situation. Notamment une toute simple se basant sur la technique hiérarchique de Havender limite le nombre de philosophes touchés à un d'un côté et deux de l'autre.

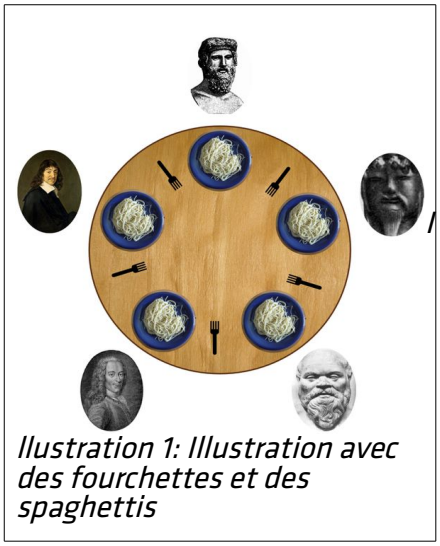


Illustration 1: Illustration avec des fourchettes et des spaghettis

Travail à faire

- En utilisant les classes fournis proposer un diagramme de classes permettant la modélisation du problème
- Mettre en oeuvre la solution