

AIAC

ACADEMIE INTERNATIONALE
MOHAMMED VI DE L'AVIATION CIVILE

DINER DES PHILOSOPHES

G20

BENCHEKROUN Mohamed
AITGOURAINE Youssef
AKSIKAS Zaid

Encadré Par : Mr KHAMMAL Adil

SOMMAIRE

- REMERCIEMENTS
- CONTEXTE ET OBJECTIFS DU TRAVAIL
- INTRODUCTION
- EXPLICATION DU CODE SOURCE
- IMPLÉMENTATION
- CONCLUSION

REMERCIEMENTS

Nous tenons tout d'abord à vous exprimer toute notre gratitude pour votre enseignement tout au long du module sur le système d'exploitation Linux. Votre passion pour le sujet, votre expertise et votre dévouement ont grandement enrichi notre expérience d'apprentissage.

Votre capacité à expliquer des concepts complexes de manière claire et accessible de plus que votre méthode d'apprentissage interactive, ont rendu ce sujet passionnant et stimulant pour nous tous. Votre soutien constant et votre encouragement ont été inestimables et ont vraiment contribué à notre épanouissement durant ce cours.

Nous sommes profondément reconnaissant d'avoir eu l'opportunité de bénéficier de vos connaissances et de votre expertise. Votre impact sur notre compréhension et notre appréciation du système d'exploitation Linux sera durable et significatif.

Merci encore pour tout ce que vous avez fait pour nous, vos étudiants. Nous vous sommes sincèrement reconnaissants et nous garderons toujours un souvenir précieux de votre cours.

BENCHEKROUN Mohamed
AITGOURAINE Youssef
AKSIKAS Zaid

CONTEXTE ET OBJECTIFS DU TRAVAIL

Imaginez une table ronde où sont assis plusieurs philosophes. Chaque philosophe a devant lui un plat, et entre chaque paire de plats, il y a une fourchette. Les philosophes passent leur temps entre deux activités : manger et réfléchir. Pour manger, un philosophe a besoin des deux fourchettes qui se trouvent de chaque côté de son assiette. Les philosophes sont des processus en parallèle qui alternent entre ces deux états.

A travers ce projet, nous allons essayer de répondre aux objectifs suivants :

Prévention des interblocages (deadlocks) : L'objectif principal est de garantir qu'aucun philosophe ne se retrouve dans une situation où il attend indéfiniment une fourchette que détient un autre philosophe, ce qui entraînerait un interblocage.

Utilisation efficace des ressources : Il est important d'optimiser l'utilisation des ressources disponibles, c'est-à-dire les fourchettes, pour maximiser le nombre de philosophes qui peuvent manger en même temps, tout en évitant les interblocages.

Équité dans l'accès aux ressources : Les philosophes devraient avoir un accès équitable aux fourchettes afin qu'aucun philosophe ne meure de faim alors que d'autres ont accès à des ressources.

Synchronisation des processus : Assurer une synchronisation adéquate entre les philosophes pour qu'ils partagent correctement les ressources (les fourchettes) et évitent les conflits.

Détection et résolution des conflits : Mettre en place un mécanisme pour détecter et résoudre les conflits potentiels qui pourraient survenir lorsque les philosophes tentent d'accéder aux fourchettes en même temps.

INTRODUCTION

LE PROBLÈME DES PHILOSOPHES DÎNANT EST UN PROBLÈME CLASSIQUE EN INFORMATIQUE QUI ILLUSTRE LES DÉFIS DE LA SYNCHRONISATION ET DE LA GESTION DES RESSOURCES PARTAGÉES DANS LES SYSTÈMES CONCURRENTIELS.

IMAGINÉ PAR EDSGER DIJKSTRA EN 1965, CE PROBLÈME MET EN SCÈNE CINQ PHILOSOPHES ASSIS AUTOUR D'UNE TABLE CIRCULAIRE, CHACUN ALTERNANT ENTRE DEUX ACTIVITÉS : PENSER ET MANGER. ENTRE CHAQUE PAIRE DE PHILOSOPHES SE TROUVE UNE UNIQUE BAGUETTE, ET POUR MANGER, UN PHILOSOPHE DOIT SAISIR LES DEUX BAGUETTES ADJACENTES.

LE DÉFI EST D'ASSURER QUE LES PHILOSOPHES PUISSENT MANGER SANS PROVOQUER DE FAMINE (OÙ UN PHILOSOPHE NE POURRAIT JAMAIS MANGER) NI DE CONDITIONS DE COURSE (OÙ PLUSIEURS PHILOSOPHES POURRAIENT SE DISPUTER UNE MÊME BAGUETTE), TOUT EN ÉVITANT LES BLOCAGES (OÙ PLUSIEURS PHILOSOPHES POURRAIENT ATTENDRE INDÉFINIMENT DES BAGUETTES TENUES PAR D'AUTRES).

EXPLICATION DU CODE SOURCE

```
CMakeLists.txt  main.c  headerfile.h x
1  #ifndef SHELL_PROJECT_HEADERFILE_H
2  #define SHELL_PROJECT_HEADERFILE_H
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <pthread.h>
6  #include <unistd.h>
7  #include <time.h>
8
9  #define N 6 // Number of philosophers
10
11  typedef struct {
12      int id; int request; pthread_cond_t condition;
13  } Philosopher;
14
15  typedef struct {
16      pthread_mutex_t mutex; pthread_cond_t server_condition;
17      int chopsticks[N]; int philosopher_requests[N];
18      Philosopher philosophers[N];
19  } Server;
20
21  Server server;
22  time_t last_meal_times[N]; // Last meal times for philosophers
23  int eating_times[N];
24  pthread_mutex_t print_mutex;
```

On commence tout d'abord par le 'headerfile.h', dans lequel on définit les bibliothèque nécessaires à notre projet, ensuite on définit le nombre de philosophes à six.

On passe ensuite à la définition des structures :

La structure Philosopher représente un philosophe avec un id, un drapeau - request) pour demander des baguettes, et une condition de synchronisation. La structure Server représente le serveur qui gère l'accès aux baguettes avec un mutex mutex, une condition de serveur server_condition, un tableau d'états des baguettes chopsticks, un tableau de requêtes des philosophes philosopher_requests, et un tableau de structures Philosopher.

Et pour comprendre ce qu'est un mutex, un mutex est un mécanisme de synchronisation utilisé dans les systèmes d'exploitation et la programmation multithread pour éviter les conditions de course lorsque plusieurs threads ou processus accèdent et manipulent des ressources partagées.

Il permet d'assurer que seulement un thread à la fois peut accéder à une section critique du code ou une ressource partagée. Cela empêche les conflits et les incohérences qui peuvent survenir lorsque plusieurs threads tentent de modifier les mêmes données en même temps.

Enfin, on trouve les variables globales, et parmi eux se trouvent :

Le serveur, les temps de repas des philosophes, le nombre de repas et un mutex pour l'impression sécurisée.

```
26 // Function prototypes
27 → void* philosopher(void* num);
28 → void* server_function(void* arg);
29 → void think(int id);
30 → void request_chopsticks(int id);
31
32 → void eat(int id);
33 → void put_down_chopsticks(int id);
34 → int check_starvation(int id);
35 → void afficher_nombre_eaten(int id);
36
37 #endif
```

Ensuite, pour pouvoir utiliser toute fonction en langage C, il nous faut impérativement inclure son prototype dans le 'headerfile.h'

```

CMakeLists.txt  main.c  headerfile.h
1  #include "headerfile.h"
2
3
4  int main() {
5      pthread_t philosophers[N];
6      pthread_mutex_init(&print_mutex, NULL);
7      pthread_t server_thread;
8      int ids[N];
9
10     // Initialize server
11     pthread_mutex_init(&server.mutex, NULL);
12     pthread_cond_init(&server.server_condition, NULL);
13     for (int i = 0; i < N; i++) {
14         server.chopsticks[i] = 1;
15         server.philosopher_requests[i] = 0;
16         pthread_cond_init(&server.philosophers[i].condition, NULL);
17         server.philosophers[i].id = i;
18         last_meal_times[i] = time(NULL); // Initialize last meal times
19     }
20     // Print starting message

```

On passe directement au fichier 'main.c', après avoir inclut notre fichier 'headerfile.h'.

Dans cet étape, on commence par créer les threads pour les philosophes, ainsi que le serveur, sans oublier l'initialisation du mutex d'impression. Le mutex du serveur et la condition du serveur sont initialisés. Chaque baguette est disponible (1), les requêtes des philosophes sont à 0, les conditions des philosophes sont initialisées, et les derniers temps de repas sont enregistrés.

```

4  int main() {
20     // Print starting message
21     pthread_mutex_lock(&print_mutex);
22     printf("=====\n");
23     printf("          DINING PHILOSOPHERS          \n");
24     printf("=====\n");
25     printf("          Number of philosophers: %d\n", N);
26     printf("=====\n");
27     pthread_mutex_unlock(&print_mutex);
28

```

Le programme imprime un message d'accueil synchronisé avec le mutex d'impression.

```

29     // Create server thread
30     pthread_create(&server_thread, NULL, server_function, NULL);
31
32     // Create philosopher threads
33     for (int i = 0; i < N; i++) {
34         ids[i] = i;
35         eating_times[i] = 0;
36         pthread_create(&philosophers[i], NULL, philosopher, &ids[i]);
37     }
38

```


Le thread du serveur est créé, puis les threads des philosophes sont créés. et pour mieux se familiariser avec cette notion de threads, Un thread est une unité d'exécution dans un programme, permettant à un programme d'effectuer plusieurs tâches simultanément. Les threads sont souvent appelés "légers" parce qu'ils partagent le même espace mémoire d'un processus, contrairement aux processus qui ont chacun leur propre espace mémoire.

```
38
39 // Join philosopher threads
40 for (int i = 0; i < N; i++) {
41     pthread_join(t: philosophers[i], res: NULL);
42 }
43
44 // Join server thread
45 pthread_join(t: server_thread, res: NULL);
46
47 // Destroy server resources
48 pthread_mutex_destroy(m: &server.mutex);
49 pthread_cond_destroy(cv: &server.server_condition);
50 for (int i = 0; i < N; i++) {
51     pthread_cond_destroy(cv: &server.philosophers[i].condition);
52 }
53 pthread_mutex_destroy(m: &print_mutex);
54
55 return 0;
56 }
```

Pour passer ensuite à l'étape d'attente, où le programme attend que tous les threads se terminent, puis enfin passer à l'étape du nettoyage, où tous les mutex et conditions sont détruits pour nettoyer correctement les ressources.

```
58 → void* philosopher(void* num) {
59     int id = *(int*)num;
60
61     while (1) {
62         think(id);
63         request_chopsticks(id);
64         eat(id);
65         put_down_chopsticks(id);
66         pthread_mutex_lock(m: &print_mutex);
67         printf(format: "Philosopher %d is sleeping\n", id+1);
68         pthread_mutex_unlock(m: &print_mutex);
69         sleep(2) ;//increase to make the starvation happen
70
71         if (check_starvation(id) == 1) {
72             exit(Code: 1);
73         }
74     }
75 }
```

On passe maintenant aux fonctions principales qui constituent notre projet, en commençant par la première fonction : 'La fonction du philosophe'

A partir de cette fonction, on peut constater que chaque philosophe pense, demandes les baguettes, mange, repose les baguettes, dort et vérifie la famine dans une boucle infinie. On sait que concernant ce projet classique le philosophe ne peut avoir que deux états, manger ou penser, cependant, nous avons décider de modifier ces états afin de complexifier un peu plus le projet.

```
77 void* server_function(void* arg) {
78     while (1) {
79         pthread_mutex_lock(&server.mutex);
80
81         for (int i = 0; i < N; i++) {
82             if (server.philosopher_requests[i] == 1) {
83                 int left = i;
84                 int right = (i + 1) % N;
85
86                 if (server.chopsticks[left] && server.chopsticks[right]) {
87                     server.chopsticks[left] = 0;
88                     server.chopsticks[right] = 0;
89                     server.philosopher_requests[i] = 0;
90                     pthread_cond_signal(&server.philosophers[i].condition);
91                 }
92             }
93         }
94
95         pthread_cond_wait(&server.server_condition, &server.mutex);
96         pthread_mutex_unlock(&server.mutex);
97     }
98 }
```

Ensuite, nous avons la fonction du serveur, qui a pour but de vérifier les requêtes des philosophes et attribue les baguettes si disponibles. Ensuite, il attend la prochaine requête.

```
100 void think(int id) {
101     int think_time = rand() % 3 + 1; // Think for a random time between 1 and 3 seconds
102     pthread_mutex_lock(&print_mutex);
103     printf("Philosopher %d is thinking\n", id+1);
104     pthread_mutex_unlock(&print_mutex);
105     sleep(think_time);
106 }
```

La prochaine fonction utilisé pour ce programme est la fonction 'think' : Cette fonction a pour but de donner aux philosophes un temps de pensée aléatoire, puis afficher un message.

```

108 → void request_chopsticks(int id) {
109     pthread_mutex_lock( m: &server.mutex);
110     server.philosopher_requests[id] = 1;
111     pthread_cond_signal( cv: &server.server_condition);
112
113     while (server.philosopher_requests[id] == 1) {
114         pthread_cond_wait( cv: &server.philosophers[id].condition, external_mutex: &server.mutex);
115     }
116     pthread_mutex_unlock( m: &server.mutex);
117 }

```

Ensuite, nous avons la fonction 'request_chopsticks', qui a pour but de demander les baguettes ainsi qu'attendre qu'elles soient disponibles.

```

119 → void eat(int id) {
120     int eat_time = rand() % 3 + 1; // Eat for a random time between 1 and 3 seconds
121     pthread_mutex_lock( m: &print_mutex);
122     printf( format: "Philosopher %d is eating \n", id+1);
123     pthread_mutex_unlock( m: &print_mutex);
124     sleep(eat_time);
125     last_meal_times[id] = time( Time: NULL); // Update last meal time
126
127     afficher_nombre_eaten(id);
128 }

```

On trouve aussi, la fonction 'eat', qui comme son nom l'indique permet au philosophe de manger pendant un temps aléatoire et met à jour son dernier temps de repas.

```

130 → void put_down_chopsticks(int id) {
131     pthread_mutex_lock( m: &server.mutex);
132     int left = id;
133     int right = (id + 1) % N;
134
135     server.chopsticks[left] = 1;
136     server.chopsticks[right] = 1;
137     pthread_cond_signal( cv: &server.server_condition);
138     pthread_mutex_unlock( m: &server.mutex);
139 }

```

La prochaine fonction utilisé pour ce programme est la fonction 'put_down_chopsticks' :

Cette fonction a pour but de permettre aux philosophes de reposer les baguettes.

```

141 → int check_starvation(int id) {
142     time_t current_time = time( Time: NULL);
143     double elapsed_time = difftime( Time1: current_time, Time2: last_meal_times[id]);
144     ⚡
145     if (elapsed_time > 1) { //decrease if you want to test the starvation
146         pthread_mutex_lock( m: &print_mutex);
147         printf( format: "Philosopher %d has starved.\n", id+1);
148         pthread_mutex_unlock( m: &print_mutex);
149         return 1;
150     }
151
152     return 0;
153 }

```

L'avant dernière fonction pour ce projet, n'est autre que la fonction 'check_starvation', qui permet aux philosophes de vérifier s'il a faim depuis trop longtemps et affiche un message s'il est affamé.

```

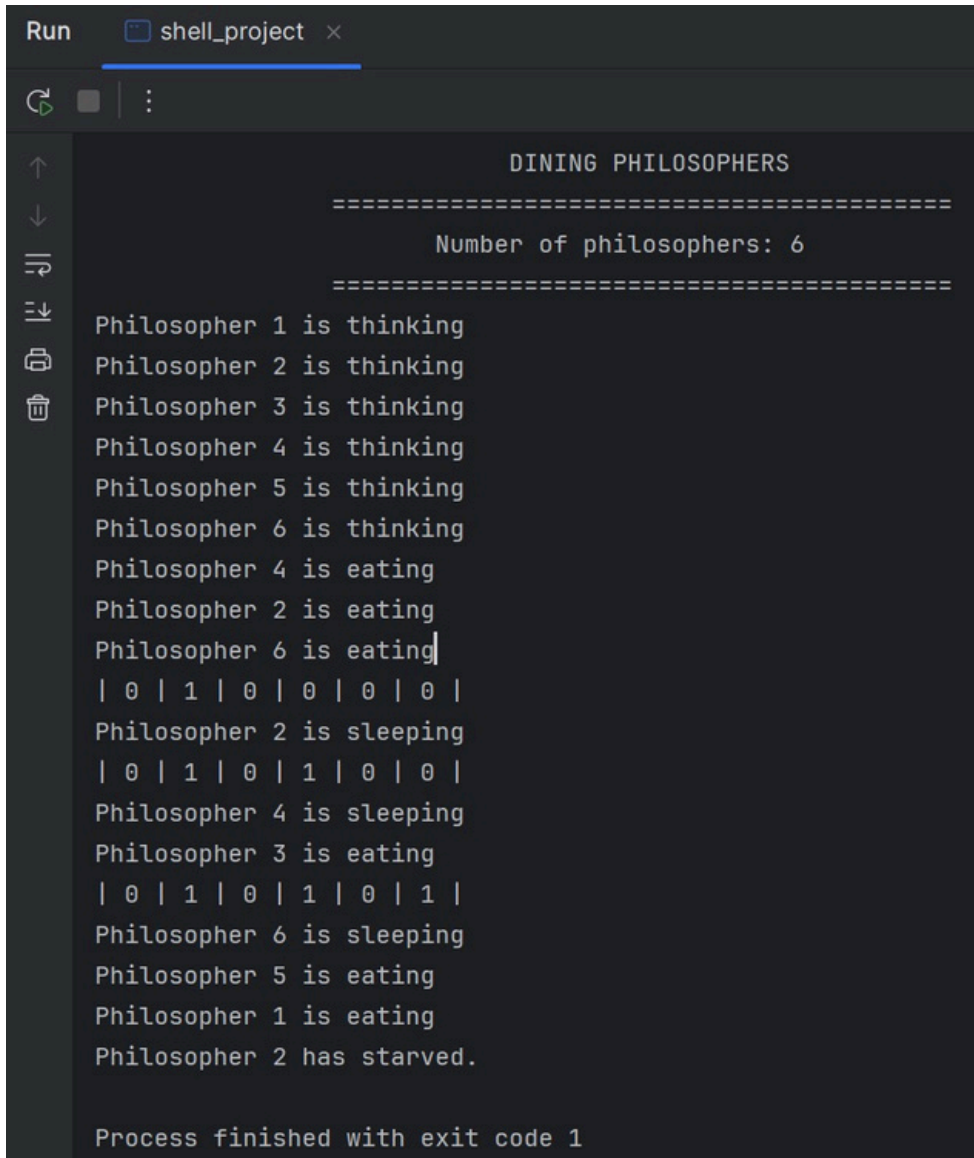
→ void afficher_nombre_eaten(int id){
    pthread_mutex_lock( m: &print_mutex);
    eating_times[id] += 1;
    printf( format: "| ");
    for (int i = 0; i < N; i++){
        printf( format: "%d | ", eating_times[i]);
    }
    printf( format: "\n");
    pthread_mutex_unlock( m: &print_mutex);
}

```

Puis finalement, pour la dernière fonction, 'afficher_nombre_eaten', celle ci permet simplement de déterminer ainsi qu'afficher le nombre de fois où chaque philosophe a manger.

IMPLÉMENTATION

Afin de mieux visualiser le résultat de ce projet, nous vous proposons d'exécuter ce programme pour voir si ce dernier fonctionne, ou s'il y a des anomalies :



```
Run  shell_project  x
=====
DINING PHILOSOPHERS
=====
Number of philosophers: 6
=====
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 6 is thinking
Philosopher 4 is eating
Philosopher 2 is eating
Philosopher 6 is eating
| 0 | 1 | 0 | 0 | 0 | 0 |
Philosopher 2 is sleeping
| 0 | 1 | 0 | 1 | 0 | 0 |
Philosopher 4 is sleeping
Philosopher 3 is eating
| 0 | 1 | 0 | 1 | 0 | 1 |
Philosopher 6 is sleeping
Philosopher 5 is eating
Philosopher 1 is eating
Philosopher 2 has starved.

Process finished with exit code 1
```

Comme on peut le voir une fois le programme exécuté, la console nous présente le nombre de philosophes concerné pour ce projet, en précisant l'état de chaque philosophe, qui permute entre penser, manger et dormir, mais le programme sort et s'arrête directement lorsqu'un philosophe atteint l'état de famine.


```
C:\Users\youuss\CLionProjects\shell_project\cmake-build-debug\shell_project.exe
=====
DINING PHILOSOPHERS
=====
Number of philosophers: 5
=====
Philosopher 1 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 2 is thinking
Philosopher 5 is thinking
Philosopher 1 is eating
Philosopher 4 is eating
| 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |
Philosopher 1 is sleeping
Philosopher 3 is eating
Philosopher 4 is sleeping
Philosopher 5 is eating
Philosopher 4 is thinking
Philosopher 1 is thinking
| 1 | 0 | 0 | 1 | 1 |
```

On peut remarquer que le nombre de philosophes n'est pas important, ce programme fonctionne pour cinq, six philosophes ou même plus.

En remédiant au problème rencontré ci-dessus, on trouve que tout fonctionne normalement, chaque philosophe pense pour un moment aléatoire, et ceux qui ont le plus faim commencent à manger, ensuite il cède les baguettes pour permettre aux autres de manger aussi.

```
Philosopher 3 is thinking
| 11 | 11 | 11 | 11 | 11 |
Philosopher 2 is sleeping
Philosopher 5 is thinking
Philosopher 4 is eating
| 11 | 11 | 11 | 12 | 11 |
Philosopher 4 is sleeping
Philosopher 1 is eating
Philosopher 2 is thinking
Philosopher 3 is eating
| 11 | 11 | 12 | 12 | 11 |
| 12 | 11 | 12 | 12 | 11 |
Philosopher 1 is sleeping
Philosopher 3 is sleeping
```

On peut constater ici que le programme peut continuer à fonctionner normalement, sans aucune anomalie, et sans arrêt, tout en affichant le nombre d'occurrence où chaque philosophe à manger.

CONCLUSION

LE PROBLÈME DES PHILOSOPHES DÎNANT ILLUSTRE PARFAITEMENT LES COMPLEXITÉS DE LA SYNCHRONISATION DANS UN ENVIRONNEMENT MULTITHREADÉ, METTANT EN LUMIÈRE LES DÉFIS TELS QUE LES BLOCAGES, LES CONDITIONS DE COURSE ET LA FAMINE.

EN UTILISANT DES MÉCANISMES DE SYNCHRONISATION COMME LES MUTEX ET LES CONDITIONS, IL EST POSSIBLE DE CONCEVOIR DES SOLUTIONS QUI PERMETTENT UNE GESTION EFFICACE DES RESSOURCES PARTAGÉES TOUT EN ÉVITANT LES ÉCUEILS COURANTS DES SYSTÈMES CONCURRENTS.

À TRAVERS CE PROBLÈME, NOUS APPRENNONS DES PRINCIPES ESSENTIELS DE LA PROGRAMMATION CONCURRENTÉ, APPLICABLES À DE NOMBREUX SCÉNARIOS RÉELS OÙ LES RESSOURCES PARTAGÉES DOIVENT ÊTRE GÉRÉES DE MANIÈRE SÛRE ET EFFICACE.

AINSI, LA MAÎTRISE DES CONCEPTS SOUS-JACENTS AU DÎNER DES PHILOSOPHES EST CRUCIALE POUR TOUT DÉVELOPPEUR TRAVAILLANT DANS DES ENVIRONNEMENTS MULTITÂCHES OU SUR DES SYSTÈMES À FORTE CONCURRENCE.

ANNEXES

VOUS ALLEZ TROUVER AVEC CE PRÉSENT RAPPORT LES CODES SOURCES DE NOTRE PROJET, AFIN DE LE CONSULTER. VOUS TROUVEREZ AUSSI UN MAKEFILE POUR LA COMPILATION DE NOTRE PROJET.