

Université [Nom de ton Université]

Département d’Informatique

Année Universitaire 2024-2025

DEVOIR N° 02

TP SYSTÈMES D’EXPLOITATION II

(Séries 04 et 05)

Réalisé par :

TADJINE Said

Groupe : B4

Chargée du module :

Dr. SABRI S.

Partie 01 : Les Threads

Question 1 : Importance et utilité du multithreading

1 Partie 01 : Les Threads (Série 04)

1.1 Question 1 : Importance et utilité du multithreading

Le multithreading est une technique essentielle dans les systèmes d'exploitation modernes. Son importance réside dans sa capacité à améliorer les performances et la réactivité des applications.

- **Parallélisme** : Il permet d'utiliser efficacement les processeurs multicœurs en exécutant plusieurs parties du code simultanément[cite : 8].
- **Réactivité** : Dans une application graphique, un thread peut gérer l'interface utilisateur pendant qu'un autre effectue des calculs lourds, évitant ainsi le gel de l'application.
- **Partage de ressources** : Contrairement aux processus, les threads d'un même processus partagent le même espace mémoire, ce qui facilite la communication et l'échange de données sans mécanismes complexes d'IPC (Inter-Process Communication).

1.2 Question 2 : Différence entre un thread et un processus

Bien que les deux représentent une exécution de code, ils diffèrent sur plusieurs points :

- **Ressources** : Un processus possède son propre espace mémoire isolé. Un thread, ou "fil d'exécution"[cite : 8], partage l'espace mémoire (code, données, tas) du processus auquel il appartient, tout en ayant sa propre pile et ses propres registres.
- **Coût** : La création et la commutation de contexte (context switch) entre threads sont beaucoup plus rapides ("légères") que celles entre processus.
- **Indépendance** : Si un thread plante, il peut entraîner l'arrêt de tout le processus. Les processus sont isolés les uns des autres.

1.3 Question 3 : Analyse du programme de la Série 04

Analyse des résultats : L'exécution du programme `thread-create.c` [cite : 25] montre un problème classique de concurrence appelé **Race Condition** (Condition de course).

- La variable `i` est globale et partagée entre le thread principal (main) et le thread fils[cite : 26].
- Les deux threads modifient `i` simultanément sans aucune synchronisation (pas de mutex ni sémaphore).
- L'ordonnancement étant asynchrone[cite : 24], l'ordre des instructions d'addition et d'affichage est imprévisible.
- **Conséquence** : Les valeurs affichées pour `i` sont incohérentes car les lectures et écritures s'entrelacent. Par exemple, le thread fils peut lire `i`, être interrompu, le thread principal modifie `i`, puis le fils reprend et écrase la modification du principal.

1.4 Exercice Supplémentaire : Programme Pair/Impair

Voici le code C réalisant l'affichage des nombres pairs et impairs par deux threads distincts, avec attente de terminaison.

Listing 1 – Programme Pair/Impair avec Threads

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <stdlib.h>
4
5 // Fonction pour le Thread 1 (Nombres pairs)
6 void *afficher_pairs(void *arg) {
7     printf("Thread 1 (Pairs):\n");
8     for (int i = 0; i <= 100; i += 2) {
9         printf("%d\n", i);
10    }
11    printf("\n");
12    pthread_exit(NULL);
13}
14
15 // Fonction pour le Thread 2 (Nombres impairs)
16 void *afficher_impairs(void *arg) {
17     printf("Thread 2 (Impairs):\n");
18     for (int i = 1; i <= 100; i += 2) {
19         printf("%d\n", i);
20    }
21    printf("\n");
22    pthread_exit(NULL);
23}
24
25 int main() {
26     pthread_t thread1, thread2;
27
28     // Cr ation des threads
29     if (pthread_create(&thread1, NULL, afficher_pairs, NULL) != 0)
30     {
31         perror("Erreur creation thread 1");
32         exit(1);
33     }
34     if (pthread_create(&thread2, NULL, afficher_impairs, NULL) != 0)
35     {
36         perror("Erreur creation thread 2");
37         exit(1);
38     }
39
40     // Attente de la fin des threads (Join)
41     pthread_join(thread1, NULL);
42     pthread_join(thread2, NULL);
43
44     printf("Fin du programme principal.\n");
45     return 0;
46 }
```

2 Partie 02 : Les Sémaphores (Série 05)

2.1 Question 4 : Problèmes liés aux sémaphores

L'utilisation incorrecte des sémaphores peut entraîner plusieurs problèmes graves dans un système :

1. **Interblocage (Deadlock)** : Une situation où deux processus ou plus s'attendent mutuellement indéfiniment. Par exemple, P1 détient la ressource A et attend B, tandis que P2 détient B et attend A.
2. **Famine (Starvation)** : Un processus peut attendre indéfiniment l'accès à une ressource si d'autres processus plus prioritaires ou plus rapides accaparent le sémaphore en permanence.
3. **Inversion de priorité** : Un processus de haute priorité est bloqué par un processus de basse priorité qui détient le sémaphore nécessaire.
4. **Erreurs de programmation** : Oublier d'effectuer une opération V (signal) après une section critique bloque définitivement les autres processus.

2.2 Question 5 : Réalisation de l'exclusion mutuelle (Mutex)

L'exclusion mutuelle permet de s'assurer qu'un seul thread accède à une section critique à la fois. Pour la réaliser avec un sémaphore :

- On utilise un sémaphore **binaire** initialisé à 1[cite : 121].
- Avant d'entrer en section critique, le processus appelle l'opération **P** (ou `wait/down`), qui décrémente le sémaphore. Si la valeur devient 0, le processus continue ; si elle est négative, il est bloqué[cite : 92].
- Après la section critique, le processus appelle l'opération **V** (ou `signal/up`), qui incrémente le sémaphore, libérant ainsi un éventuel processus en attente.

2.3 Exercice Supplémentaire : Producteur-Consommateur

Voici le pseudo-code résolvant le problème du Producteur-Consommateur à l'aide de sémaphores pour gérer le tampon (buffer) borné.

Listing 2 – Pseudo-code Producteur-Consommateur

```
1 // Initialisation
2 Semaphore mutex = 1;           // Exclusion mutuelle pour l'accès au
3 Semaphore vide = N;            // Places vides dans le tampon (taille N)
4 Semaphore plein = 0;           // Places occupées (items produits)
5
6 Producteur() {
7     while (VRAI) {
8         item = produire_item();
9
10        P(vide);    // Attendre qu'il y ait une place libre
11        P(mutex);   // Verrouiller l'accès au tampon
12
13        déposer_item(item); // Section Critique
14    }
}
```

```
15     V(mutex); // Deverrouiller le tampon
16     V(plein); // Signaler qu'il y a un nouvel item
17 }
18 }
19
20 Consommateur() {
21     while (VRAI) {
22         P(plein); // Attendre qu'il y ait un item a consommer
23         P(mutex); // Verrouiller l'acc s au tampon
24
25         item = retirer_item(); // Section Critique
26
27         V(mutex); // Deverrouiller le tampon
28         V(vide); // Signaler qu'une place s'est liberee
29
30         consommer_item(item);
31     }
32 }
```