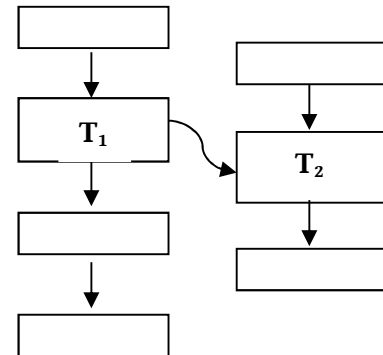


TRAVAUX PRATIQUES SEMAPHORES

EXERCICE 1 - Précédence de tâches

Un système est composé de deux tâches (traitements) T_1 et T_2 soumises à la contrainte de précédence $T_1 < T_2$. Ces deux tâches appartiennent à deux processus différents qui doivent être synchronisés \Rightarrow **Le deuxième processus doit retarder l'exécution de la tâche T_2 jusqu'à ce que le premier processus termine la tâche T_1 .**



EXERCICE 2 - Rendez-vous à 2 et à 3

Deux processus **P1** et **P2** souhaitent établir un rendez-vous avant l'exécution de la fonction `fRendezVous_1()` pour l'un et `fRendezVous_2()` pour l'autre. En utilisant les sémaphores, écrire les programmes **P1.c** et **P2.c** permettant d'établir ce rendez-vous.

*Version 2 : Rendez-vous à 2 et à 3 : Réaliser un rendez-vous entre 3 processus **P1**, **P2** et **P3**.*

EXERCICE 3 - Le Rendez-vous Emetteurs/Récepteurs

Pour réaliser un mécanisme de communication un à plusieurs, on utilise un ensemble de processus composé d'émetteurs et de récepteurs. Un émetteur produit un message (**à simuler par l'affichage d'un message sur écran**) et se met en attente jusqu'à ce qu'il y ait **n-1** récepteurs au rendez-vous. Un récepteur lancé attend l'émission et l'arrivée des autres récepteurs. Prenons l'exemple d'un rendez-vous **1 à 2** :

- | | |
|----------------------|--|
| 1. \$> recepteur & | // Le récepteur 1 se met en attente |
| 2. \$> emmetteur i & | // L'émetteur 2 affiche le message i et se met en attente |
| 3. \$> emmetteur j & | // L'émetteur 3 produit le message j et se met en attente |
| 4. \$> recepteur & | // Le récepteur 4 débloquent 1 et 2. |
| 5. \$> recepteur & | // Le récepteur 5 au Rendez-vous avec l'émetteur 3 - Attente |
| 6. \$> recepteur & | // Débloquent 3 et 5. |

Un processus **prod** produit des informations qui sont consommées par un autre processus **conso**. La communication entre les processus se fait par l'intermédiaire d'un segment de mémoire partagée nommé **buffer** de taille **N** fixée suivant les règles suivantes :

- **prod** dépose des informations dans le **buffer**. Ce processus doit attendre si **buffer** est plein.
- **conso** récupère les informations stockées dans **buffer**. Ce processus doit attendre si **buffer** est vide.
- Le segment mémoire **buffer** est une ressource critique (partagée par plusieurs processus).
- Les informations sont consommées dans l'ordre où elles sont produites [Premier Arrivé/Premier Consommé - FIFO].
- Il peut y avoir plusieurs **prod** et **conso** partageant la ressource **buffer**.

Réalisez une implémentation de ce modèle dans l'environnement *Linux* sous forme de 4 programmes exécutables :

- **init** créé et initialise les ressources système partagées (Buffer, sémaphores, ...).
- **prod** produit un caractère dans **buffer**. Ce caractère est passé en paramètre de la ligne de commande.
- **conso** consomme un caractère du **buffer**.
- **clear** supprime les ressources partagées.

Exemple : pour lancer **3 conso** et **2 prod**, vous pouvez utiliser la ligne de commande avec la mise en arrière-plan :

\$ conso & prod a & prod b & conso & conso &

L'objectif de cet exercice est de créer un ensemble de **N** serveurs (processus) lancés indépendamment par l'intermédiaire d'une commande sans paramètre de nom **serveur** (lancer **N** fois la commande serveur). Ces **N** processus doivent communiquer par l'intermédiaire d'une mémoire partagée. Chaque processus doit être identifié par un numéro **id** (un entier) indépendant de son **pid**.

Ecrire le code C correspondant à la commande **serveur**. Un protocole particulier devra permettre à chaque processus, d'une part de s'attribuer un identificateur à sa création et d'autre part d'attendre, avant de se recouvrir par une commande **cmd**, dépendant de l'identificateur **id** du processus, que les **N** processus existent.

Les contraintes suivantes doivent être respectées :

- ✓ Les mécanismes de communication et de synchronisation utilisés seront les **IPC**.
- ✓ Les **IPC** utilisé(s) par le protocole d'attribution des identificateurs sont tous supprimés une fois les processus créés.
- ✓ A un instant donné, il ne pourra exister qu'un seul système de **N** processus de ce type.
- ✓ La solution ne doit comporter aucune attente active.

Deux processus **P1** et **P2** disposant chacun d'un tableau stockant **N** entiers. **P1** et **P2** veulent pouvoir s'échanger des nombres de leur tableau de manière à ce que, à la fin de leur exécution réciproque, **P1** dispose des **N** plus petites valeurs et **P2** des **N** plus grandes.

Besoins : Mémoire partagée composée de trois cases (3 entiers) + un processus coordinateur **P0**.

Fonctionnement

- **P1** cherche le **maximum** dans son tableau et l'écrit dans la 1^{ère} case de la mémoire partagée.
- **P2** cherche le **minimum** dans son tableau et l'écrit dans la 2^{ème} case de la mémoire partagée.
- **P0** vérifie que la valeur de la première case est supérieure à la valeur de la seconde case. Si c'est le cas, **P0** effectue l'échange des valeurs, sinon il met la valeur **-1** dans la troisième case de la mémoire partagée.
- Chaque processus (**P1** et **P2**) récupère sa nouvelle valeur dans la case respective, l'écrit à la place de l'ancienne, puis recommence le même travail.

Si la valeur de la troisième case est égale à **-1**, les processus n'effectuent aucun échange. Ils affichent leur tableau et s'arrêtent.

Les fonctions à développer :

- **int mini(int table[])**
Retourne la position (indice) du plus petit élément du tableau.
- **int maxi(int table[])**
Retourne la position (indice) du plus grand élément du tableau.
- **void echange(int table[], int i , int j)**
Permute les deux éléments se trouvant aux indices **i** et **j**.

En utilisant les sémaphores et la mémoire partagée, écrivez les programmes de **P0.c**, **P1.c** et **P2.c**.
Donnez les valeurs initiales des variables partagées et des sémaphores.

Les trois processus doivent se terminer lorsqu'il n'y a plus de données à échanger.

Annexe : Implémentation des sémaphores de DIJKSTRA

Le code **C** ci-dessous réalise l'implémentation des sémaphores de **Dijkstra** à partir des mécanismes de sémaphores. La fonction **sem_create()** permet de créer un sémaphore. Les opérations **P** et **V** sont réalisées par les fonctions **P()** et **V()**. La fonction **sem_delete()** permet de détruire un sémaphore.

```
int sem_create(key_t cle, int initval) {
    int semid ;
    union semun {
        int val ;
        struct semid_ds *buf ;
        ushort *array ;
    } arg_ctl ;
    semid = semget(cle,1,IPC_CREAT | IPC_EXCL | 0666);
    if (semid == -1) {
        semid = semget(cle,1, 0666);
        if (semid == -1) {
            perror("Erreur semget()") ;
            exit(1) ;
        }
    }
    else {
        arg_ctl.val = initval ;
        if (semctl(semid,0,SETVAL,arg_ctl) ==-1)
        {
            perror("Erreur semctl ") ;
            exit(1) ;
        }
    }
    return(semid) ;
}

void P(int semid) {
    struct sembuf sempar ;
    sempar.sem_num = 0 ;
    sempar.sem_op = -1 ;
    sempar.sem_flg = 0 ;
    if (semop(semid, &sempar, 1) == -1)
        perror("Erreur operation P") ;
}

void V(int semid) {
    struct sembuf sempar ;
    sempar.sem_num = 0 ;
    sempar.sem_op = 1 ;
    sempar.sem_flg = 0 ;
    if (semop(semid, &sempar, 1) == -1)
        perror("Erreur opération V") ;
}

void sem_delete(int semid) {
    if (semctl(semid,0,IPC_RMID,0) == -1)
        perror("Erreur semctl ") ;
}
```