TP2_SE_L2

Rappelle :

- La commande fork() est utilisée pour créer un nouveau processus appelé : « processus fils » à partir du processus appelant, également appelé « processus père ».
- Le processus fils est une **copie identique du processus père**, avec le même programme en cours d'exécution.
- Le processus fils commence son exécution à partir de l'instruction fork() et n'exécute pas les instructions précédant cette instruction.
- La fonction fork() retourne une valeur différente en fonction du processus qui l'exécute :
  - Si la fonction est exécutée par le processus père, elle renvoie : **le numéro (pid) du processus fils.**
  - Si la fonction est exécutée par le processus fils, elle renvoie : **la valeur 0**
    - Remarque : le pid du processus père peut être obtenu à l'aide de la fonction getppid().
  - Dans le cas où la table des processus est pleine et qu'aucun nouveau processus ne peut être créé, la fonction fork() renvoie : **la valeur -1.**

Exercice 1 :

Q1) Ecrire sur votre éditeur le programme ci-dessous et donner son schéma d'exécution :

```
#include <stdio.h>

#include <unistd.h>

int main() {

        fork();
        printf("Hello \n") ;

        return 0 ;

}
```

Q2) refaire le même travaille avec les 3 programmes suivants :

```
#include <stdio.h>
#include <unistd.h>
int main() {
        printf("Avant fork \n") ;
        fork();
        printf("Apres fork\n") ;
        return 0 ;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main() {
        fork();
        fork();
        printf("Hello \n") ;
        return 0 ;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main() {
        fork();
        fork();
        fork();
        printf("Hello \n") ;
        return 0 ;
}
```

Q3) On considère les 4 programmes ci-dessous.

Programme 1 :

```
1  int main() {
2    int i;
3
4    for (i=0; i<2; i++)
5      fork();
6    printf("hello!\n");
7    exit(0);
8  }
```

Programme 2 :

```
1  void doit() {
2    fork();
3    fork();
4    printf("hello!\n");
5  }
6  int main() {
7    doit();
8    printf("hello!\n");
9    exit(0);
10 }
```

Programme 3 :

```
1  int main() {
2    if (fork())
3      fork();
4    printf("hello!\n");
5    exit(0);
6  }
```

Programme 4 :

```
1  int main() {
2    if (fork()==0) {
3      if (fork()) {
4        printf("hello!\n");
5      }
6    }
7  }
```

- Combien de lignes hello ! Imprime chacun des programmes suivants ?
- Comment on peut déduire la différence entre le père et le fils dans un petit programme ?

Exercice 2 : Commutation du contexte/ gestion des priorités :

Q1) Écrire un programme père qui crée trois fils où :

- Chacun des fils doit exécuter une boucle de 100 itérations.
- À chaque itération, chaque fils doit afficher un message indiquant qu'il s'agit d'un fils, en indiquant son PID et le numéro de l'itération (valeur de i de la boucle).
- Le père doit également afficher un message indiquant qu'il s'agit du processus père.

Q2) En observant les résultats affichés, qu'est-ce que vous pouvez déduire ?

Exercice 3 :

Voici le code suivant :

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

#define MEM_SIZE 300   // Taille totale de la mémoire
#define MAX_PROGS 5    // Nombre maximum de programmes pouvant être alloués
#define FREE 0         // Marqueur de partition libre
#define OCCUPIED 1     // Marqueur de partition occupée

// Structure pour représenter une partition de mémoire
typedef struct Partition {
    int start;          // Adresse de début de la partition
    int size;           // Taille de la partition
    int status;         // Statut de la partition (libre ou occupée)
} Partition;

Partition memory[MEM_SIZE];     // Tableau pour représenter la mémoire
int num_progs = 0;              // Nombre de programmes alloués

// Initialise la mémoire en créant une partition libre de taille MEM_SIZE
void init_memory() {
    memory[0].start = 0;
    memory[0].size = MEM_SIZE;
    memory[0].status = FREE;
}

// Alloue une partition de taille size dans la mémoire en utilisant la méthode First Fit
void allocate_first_fit(int size) {
    int i;
    for (i = 0; i < MEM_SIZE; i++) {
        if (memory[i].status == FREE && memory[i].size >= size) {
            // Partition libre de taille suffisante trouvée
            memory[i].status = OCCUPIED;
            if (memory[i].size > size) {
                // Créer une nouvelle partition libre pour l'espace restant
                memory[i+1].start = memory[i].start + size;
                memory[i+1].size = memory[i].size - size;
                memory[i+1].status = FREE;
            }
            memory[i].size = size;
            printf("Programme de taille %d alloué à l'adresse %d\n", size, memory[i].start);
            num_progs++;
            return;
        }
    }
    // Partition libre de taille suffisante introuvable
    printf("Erreur : Pas assez d'espace pour allouer un programme de taille %d\n", size);
}

int main() {
    // Initialise la mémoire
    init_memory();

    // Alloue les programmes dans l'ordre demandé
    allocate_first_fit(80);
    allocate_first_fit(42);
    allocate_first_fit(95);
    allocate_first_fit(35);
    allocate_first_fit(50);

    return 0;
}
```

- Recopie le code et exécuter le
- qu'est ce qu'il fait ce programme ?
- donner la trace d'exécution.