TP3 — Listes Chaînées

NF16 – Algorithmique et Structures de Données

Automne 2025

Contexte

Dans un système d'exploitation, **l'ordonnancement des processus** consiste à décider quel processus utilisera le processeur à un instant donné. Différentes **politiques d'ordonnancement** existent, chacune visant un objectif : simplicité, équité, ou optimisation du temps de réponse.

L'objectif de ce TP est de manipuler **deux variantes de listes chaînées** : (i) File FIFO et (ii) Liste triée. Dans ce TP, nous simulerons **deux politiques** : (i) FCFS et (ii) SJF.

— **First-Come, First-Served (FCFS)** : les processus sont exécutés dans l'ordre de leur arrivée, sans préemption. – ***Structure de données adaptée :*** *File FIFO où l'on insère les processus à la fin et l'on retire au début.*

— **Shortest Job First (SJF)** : les processus ayant la plus courte durée sont exécutés en priorité. On considère une version sans préemption (on laisse finir le processus en cours). – ***Structure de données adaptée :*** *Liste triée par durée. Chaque nouvelle arrivée est insérée dans la liste à sa position correcte pour conserver l'ordre croissant.*

Objectifs pédagogiques

— Implémenter et maîtriser deux structures : **file FIFO** et **liste triée**.

— Réaliser les opérations de base : **insertion**, **retrait**, **parcours**, **libération**, etc.

— Simuler deux politiques d'ordonnancement : **FCFS** et **SJF**.

Spécifications et fonctions à implémenter

A. Structures de données

Q1. Définir la structure et le type correspondant aux processus comme suit :

```c
typedef struct s_processus{
    int pid;                        // identifiant du processus
    int arrivee;                    // temps d'arrivee
    int duree;                      // duree d'execution
    struct s_processus *suivant;    // pointeur vers le processus suivant
} t_processus;
```

B. Fonctions communes

Q2. Implémenter les fonctions qui permettent de : (i) créer un nouveau processus et renvoyer un pointeur sur ce dernier et (ii) supprimer un processus.

```c
t_processus* creer_processus(int pid, int arrivee, int duree);
void free_processus(t_processus* p);
```

C. Fonction pour le chargement depuis un fichier

Q3. Implémenter une fonction qui charge depuis un fichier texte la liste des processus dans un tableau alloué dynamiquement de structures `t_processus`, par mesure de simplification.

```
t_processus* charger_processus(char* nom_fichier, int* nb_processus);
```

Fichier d'entrée :

```
pid arrivee duree
1   0       5
2   2       2
3   4       1
```

Le fichier fourni ne comporte pas d'en-tête. Les champs sont séparés par des espaces. Il est trié selon le temps d'arrivée (ordre croissant).

D. Fonctions pour l'ordonnancement FCFS (file FIFO)

Q4. Implémenter les fonctions qui permettent de : (i) créer une file vide et (ii) libérer la mémoire d'une file.

```
t_processus* fifo_init ();
t_processus* fifo_clear (t_processus* file);
```

Q5. Implémenter la fonction qui permet de vérifier si une file est vide.

```
int fifo_vide (t_processus* file);  // retourne 1 si la liste est vide, 0 sinon
```

Q6. Implémenter les fonctions qui permettent de : (i) enfiler un élément dans la file (ii) défiler un élément de la file.

```
t_processus* fifo_enfiler (t_processus* file, t_processus* p);
t_processus* fifo_defiler (t_processus** file);
```

Q7. Implémenter la simulation d'ordonnancement FCFS, décrite comme suit :

```
void simuler_fcfs(t_processus* tableau, int nb_processus);
```

Chargement des arrivées. À chaque tick d'horloge t, admettre tous les processus dont `arrivee == t` dans la file d'attente. Pour deux processus arrivant au même tick, celui placé avant dans le tableau sera servi en premier.

FCFS. L'ordonnancement se déroule comme suit, tant qu'il reste des processus à exécuter :

1. Si le processeur est libre et que la file n'est pas vide, retirer le processus en tête de file et l'exécuter **jusqu'à son achèvement complet**.
2. À chaque tick, parallèlement à l'exécution, continuer d'admettre les nouveaux processus arrivés dans la file via `fifo_enfiler`.

Comme la politique est **non préemptive**, un processus déjà en cours d'exécution n'est jamais interrompu, même si de nouveaux processus arrivent dans la file.

Trace d'exécution.

```
=== Simulation FCFS ===
t=0 : arrivee P1 (duree=5)
t=0 : run P1 (duree=5)
t=2 : arrivee P2 (duree=2)
t=4 : arrivee P3 (duree=1)
```

```
t=5 : fin P1
t=5 : run P2 (duree=2)
t=7 : fin P2
t=7 : run P3 (duree=1)
t=8 : fin P3
```

E. Fonctions pour l'ordonnancement SJF (liste triée)

Q8. Implémenter les fonctions qui permettent de : (i) créer une liste vide et (ii) libérer la mémoire d'une liste.

```
t_processus* lsorted_init();
t_processus* lsorted_clear(t_processus* liste);
```

Q9. Implémenter la fonction qui permet de vérifier si une liste est vide.

```
int lsorted_vide(t_processus* liste); // retourne 1 si la liste est vide, 0 sinon
```

Q10. Implémenter la fonction qui permet d'insérer un nouveau processus dans la liste triée par la durée d'exécution, dans l'ordre croissant, à la bonne position. En cas d'égalité de durée, le nouvel élément est inséré après les éléments existants de même durée.

```
t_processus* lsorted_inserer_trie(t_processus* liste, t_processus* p);
```

Q11. Implémenter la fonction qui permet de récupérer l'élément en tête de liste.

```
t_processus* lsorted_extraire_premier(t_processus** liste);
```

Q12. Implémenter la simulation d'ordonnancement SJF, décrite comme suit :

```
void simuler_sjf(t_processus* tableau, int nb_processus);
```

Chargement des arrivées. À chaque tick d'horloge t, admettre tous les processus dont `arrivee == t` dans la liste. Chacun de ces processus est inséré dans la liste des processus prêts, de manière ordonnée par durée d'exécution croissante.

SJF (non préemptif). L'ordonnancement fonctionne comme suit, tant qu'il reste des processus à exécuter :

1. Si le processeur est libre et que la liste n'est pas vide, retirer le premier élément de la liste et exécuter ce processus **jusqu'à son achèvement complet**.
2. Lorsqu'un nouveau processus arrive, l'insérer dans la liste triée des processus prêts.

La politique est **non préemptive** : un processus en cours ne peut pas être interrompu par l'arrivée d'un autre, même plus court.

Trace d'exécution.

```
=== Simulation SJF ===
t=0 : arrivee P1 (duree=5)
t=0 : run P1 (durée=5)
t=2 : arrivee P2 (duree=2)
t=4 : arrivee P3 (duree=1)
t=5 : fin P1
t=5 : run P3 (duree=1)
t=6 : fin P3
t=6 : run P2 (duree=2)
t=8 : fin P2
```

E. Interface (menu)

Q13. Implémenter la fonction principale, en se basant sur le code fourni. Il faudra compléter le `case` correspondant au chargement des processus (pensez à libérer l'espace mémoire du tableau avant rechargement).

```c
int main (){
    t_processus* tableau = NULL; // tableau de processus
    int nb_processus = 0;
    int choix;
    char nom_fichier[256];

    do {
        printf("\n===== Menu =====\n");
        printf("1. Charger processus depuis un fichier\n");
        printf("2. Simuler FCFS\n");
        printf("3. Simuler SJF\n");
        printf("4. Quitter\n");
        printf("Votre choix : ");
        scanf("%d", &choix);

        switch (choix) {
            case 1: // Charger processus
                //A COMPLETER
                break;

            case 2: // Simuler FCFS
                if (tableau == NULL) {
                    printf("Veuillez charger un fichier d'abord.\n");
                } else {
                    simuler_fcfs(tableau, nb_processus);
                }
                break;

            case 3: // Simuler SJF
                if (tableau == NULL) {
                    printf("Veuillez charger un fichier d'abord.\n");
                } else {
                    simuler_sjf(tableau, nb_processus);
                }
                break;

            case 4: // Quitter
                printf("Au revoir !\n");
                break;

            default:
                printf("Choix invalide.\n");
        }

    } while (choix != 4);

    // Nettoyage mémoire avant sortie
    if (tableau != NULL) {
        free(tableau);
    }

    return 0;
}
```

Menu :

```
1. Charger processus depuis un fichier
2. Simuler FCFS
3. Simuler SJF
4. Quitter
```

Consignes

— À la fin de l'exécution, **tous les blocs de mémoire alloués dynamiquement doivent être correctement libérés**, afin d'éviter toute fuite mémoire.

— Vous veillerez à **optimiser vos algorithmes**, tant sur le plan de la complexité que de la clarté du code.

— L'organisation minimale du projet est la suivante : un fichier d'en-tête `tp3.h`, contenant la déclaration des structures et des fonctions de base ; un fichier source `tp3.c`, regroupant l'implémentation de ces fonctions ; un fichier source `main.c`, contenant le programme principal.

— Le rapport (quatre pages maximum) devra présenter : une **analyse de la complexité** de chacune des fonctions implémentées ; la **liste des structures et fonctions supplémentaires** que vous avez choisies d'ajouter, ainsi que la justification de ces choix.

— Le rendu comprendra le rapport ainsi que les trois fichiers sources. L'ensemble devra être déposé sur Moodle, dans l'espace dédié, après la démonstration devant le chargé de TP. Un seul dépôt par binôme est attendu.

Annexe : Visualisation des ordonnancements

Cette annexe illustre, sur un exemple simple, la différence de comportement entre les deux politiques d'ordonnancement étudiées : *FCFS* et *SJF* en version non préemptive. Nous utiliserons les données du fichier exemple donné ci-haut.

Résultats de la simulation. Les traces montrent que :

— En **FCFS**, les processus sont exécutés strictement dans l'ordre d'arrivée. P1 occupe le processeur de $t = 0$ à $t = 5$, puis P2 de $t = 5$ à $t = 7$, et enfin P3 de $t = 7$ à $t = 8$.

— En **SJF**, le choix se fait en fonction de la plus courte durée parmi les processus disponibles. Après la fin de P1 à $t = 5$, c'est P3 (durée 1) qui est choisi avant P2 (durée 2), réduisant ainsi le temps moyen d'attente.

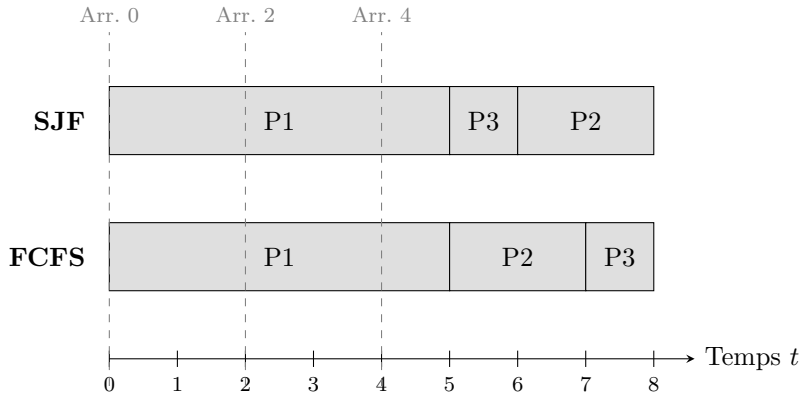

FIGURE 1 – Diagramme de Gantt comparatif : FCFS vs. SJF (non préemptif).

Autre exemple.

```
pid arrivee duree
1 0 5
2 1 3
3 2 8
4 3 6
5 4 2
```

```
=== Simulation FCFS ===
t=0 : arrivee P1 (duree=5)
t=0 : run P1 (duree=5)
t=1 : arrivee P2 (duree=3)
t=2 : arrivee P3 (duree=8)
t=3 : arrivee P4 (duree=6)
t=4 : arrivee P5 (duree=2)
t=5 : fin P1
t=5 : run P2 (duree=3)
t=8 : fin P2
t=8 : run P3 (duree=8)
t=16 : fin P3
t=16 : run P4 (duree=6)
t=22 : fin P4
t=22 : run P5 (duree=2)
t=24 : fin P5
```

```
=== Simulation SJF ===
t=0 : arrivee P1 (duree=5)
t=0 : run P1 (duree=5)
t=1 : arrivee P2 (duree=3)
t=2 : arrivee P3 (duree=8)
t=3 : arrivee P4 (duree=6)
t=4 : arrivee P5 (duree=2)
t=5 : fin P1
t=5 : run P5 (duree=2)
t=7 : fin P5
t=7 : run P2 (duree=3)
t=10 : fin P2
t=10 : run P4 (duree=6)
t=16 : fin P4
t=16 : run P3 (duree=8)
t=24 : fin P3
```