Listes chaînées

Description du problème

Il s'agit de développer une application simplifiée pour la gestion d'une série de tâches à accomplir par un processeur appelé « Colombo ».
Pour l'instant chaque tâche est caractérisée par :

- Un identifiant unique
- Une durée (en période processeur)

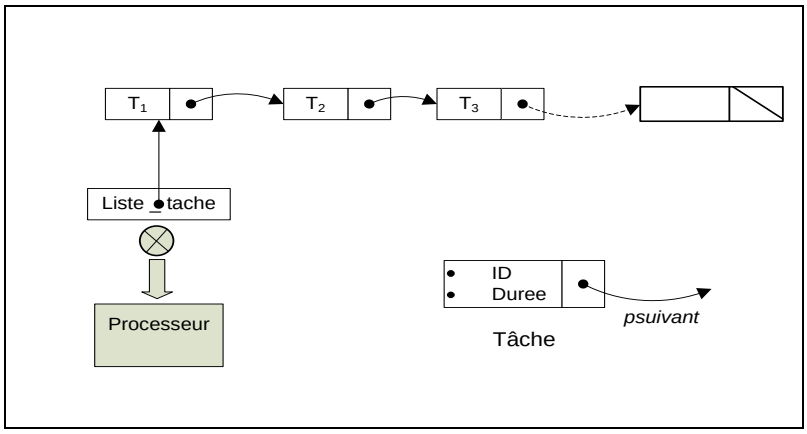

Figure 1 Représentation du problème

Le processeur «Colombo», *monocore*, doit gérer un ensemble de tâches $\mathbf{T_i}$ qu'il reçoit en permanence de différents processus s'exécutant sur la machine. Nous supposons que son fonctionnement n'est pas préemptif, c'est-à-dire qu'il ne peut pas arrêter le traitement d'une tâche dont il a commencé l'exécution au profit d'une autre. La liste des tâches est identifiée par un pointeur (*liste_tache*) pointant sur la première tâche à exécuter.

Nous allons ainsi définir une structure de tâches reçues par le processeur de la manière suivante :

```
typedef struct task
{
      char ID[MAX_NOM+1];      //Identifiant (nom) unique de la tâche
      int duree;               //Durée de la tâche (en période processeur)
      task *psuivant;          //Pointeur à la tâche suivante
}task;
```

Dans la liste des tâches, chacune contient un pointeur vers la tâche suivante à exécuter. Le pointeur de la dernière tâche de la liste est égal à NULL. Chaque nouvelle tâche reçue sera ajoutée en fin de la liste.

Options : La liste pourra être gérée à l'aide d'une sentinelle

Fonctions à réaliser

Créer les fonctions de bases suivantes pour la manipulation des listes chaînées :

1. `task * cree_tache(char caract[MAX_NOM+1], int duree)` : Fonction renvoyant un pointeur de type *task* sur une tâche d'identifiant "`caract`" et de durée "`duree`"
2. `task * cree_liste(task *tache)`: Fonction initialisant une liste à partir d'un pointeur de tâche "`tache`"
3. `void affiche_liste(task *list_task)` : Fonction affichant une liste de tâches
4. `int ajoute_tache(task *list_task, task *ptache)` : Fonction ajoutant une nouvelle tâche "`ptache`" à la fin de la liste des tâches "`list_task`"
5. `task * annule_tache(task *list_task, char caract[MAX_NOM+1])` : Un processus ayant envoyé une tâche d'identifiant "`caract`" a la possibilité d'annuler l'exécution de cette tâche par le processeur avant que ce dernier ne l'ait traitée
6. `task * libere_liste(task *list_task)`: Fonction qui libère l'espace mémoire allouée pour les différents éléments de la liste des tâches.

Dans ce qui suit, nous allons considérer différents types de fonctionnement du processeur.

Fonctionnement du processeur

Fonctionnement en mode FIFO

L'acronyme FIFO, abréviation de *First In, first Out*, que l'on peut traduire par « premier arrivé, premier sorti », est un terme souvent employé pour décrire une méthode de traitement des éléments d'une file d'attente (calculs d'un ordinateur, stocks).

Le processeur fonctionnant en mode FIFO exécutera ainsi les tâches dans l'ordre de leur arrivée. La fonction suivante sera implémentée et testée.

7. `task * execute_tache_FIFO(task *list_task);`

Fonctionnement en mode LIFO

L'acronyme LIFO, *Last In, First Out*, signifie « dernier arrivé, premier sorti ». La dernière donnée ajoutée à la structure sera ainsi la première à être retirée. La structure de pile repose sur ce principe. Ainsi une tâche reçue en dernier sera la première traitée. La fonction suivante sera implémentée et testée.

8. `task * execute_tache_LIFO(task *list_task);`

Fonctionnement en mode FIFO, avec priorité selon la durée des tâches

Dans cette partie, on retourne au cas d'un processeur fonctionnant en mode FIFO. Mais lors de la réception d'une tâche, celle-ci devra être insérée dans la liste, en respectant un ordre croissant de la durée des tâches (champ "`duree`"). Ainsi des tâches de durée courte seront prioritaires à l'exécution par rapport à des tâches longues.

Une fonction supplémentaire (fonction `insere_tache`) doit être programmée. L'exécution des tâches sera faite avec la fonction `execute_tache_FIFO` utilisée précédemment.

```
9. task * execute_tache_FIFO(task *list_task);
```

Pour tester le bon fonctionnement de ce mode, les tâches seront chargées à partir du fichier texte *tasks.dat*. La structure des données de ce fichier est (pour chaque ligne) :

nom_tache\t*duree_tache*\n

Ainsi, pour lire ce fichier la fonction *fscanf* sera utilisée :

```
fscanf(fsource,"%s\t%d\n",nom,&duree);
```

Pour simuler un fonctionnement réel, les 10 premières tâches seront tout d'abord chargées dans la liste des tâches. Ensuite, à chaque pas de temps, le programme devra charger une nouvelle tâche et en exécuter une de la liste.

Il est également demandé de développer une fonction qui fusionne deux listes de tâches, en respectant toujours l'ordre croissant de leur durée d'exécution.

```
10. task * fusion_listes(task *list_task1, task *list_task2);
```

Quels sont les avantages et inconvénients de ce mode de fonctionnement intégrant la notion de priorité selon la durée d'exécution des tâches ?

Fonctionnement en mode FIFO, avec changement dynamique de priorité

Pour illustrer les problèmes liés au fait de privilégier l'exécution de tâches de courtes durées, il est demandé d'utiliser les tâches du fichier *tasksf.dat*. Comme précédemment, les 10 premières tâches seront tout d'abord chargées dans la liste. Ensuite, à chaque pas de temps, le programme devra charger une nouvelle tâche et en exécuter une de la liste.

Montrer le problème de " *famine* " qui apparaît. Pour palier à ce problème, nous allons faire évoluer la structure de tâche en y ajoutant un champ priorité. Celle-ci devient alors :

```
typedef struct task
{
      char ID[MAX_NOM+1];     //Identifiant (nom) unique de la tâche
      int duree;              //Durée de la tâche (en période processeur)
      int priorité;           //Priorité de la tâche
      task *psuivant;         //Pointeur à la tâche suivante
}task;
```

Les valeurs du champ priorité sont comprises entre 1 et 5. Elles sont initialisées proportionnellement à la durée des tâches, en divisant la valeur du champ "*durée*" par 10 :

priorite = maximum (5, duree/10)

A chaque exécution d'une tâche de la liste, le champ priorité de l'ensemble de toutes les tâches présentes sera décrémenté d'une unité.

Chaque nouvelle tâche qui arrive sera insérée selon l'ordre croissant du champ priorité. Ainsi, des tâches dont le champ priorité est bas seront privilégiées à l'exécution.

Il est ainsi demandé de développer deux fonctions supplémentaires :

11. `task * insere_tache_priorite(task *list_task, task *ptache)` : Fonction qui insère la tâche " *ptache* " dans " *list_task*" selon l'ordre croissant du champ priorité

12. `int MAJ_priorite(task *list_task)` : Fonction qui Met à Jour le champ priorité de l'ensemble des tâches de la liste.

Comme précédemment, pour simuler un fonctionnement réel, les 10 premières tâches seront tout d'abord chargées (du fichier *tasksf.dat*) dans la liste des tâches. Ensuite, à chaque pas de temps, le programme devra charger une nouvelle tâche et en exécuter une de la liste.

Remarques Générales

- Un menu utilisateur permettant de lancer toutes les fonctions développées pendant ce TP, devra être programmé

- L'exécution des tâches par le processeur sera simulée par la fonction *Sleep()* de la librairie "`windows.h`"

- D'autres fonctions seront nécessaires au bon fonctionnement de l'ensemble des modes de traitement demandés. A vous de les identifier et de les développer

- Un rapport d'au plus 4 pages sera à rendre, contenant une introduction, les choix de programmation, le calcul de complexité des principaux algorithmes et une conclusion. **Pas de code source dans le rapport**.

- Le programme doit être composé d'au moins 3 fichiers :

  tp3.h : contient les constantes globales, les types définis et les prototypes de toutes les fonctions. Chaque fonction doit être déclarée avec un commentaire en expliquant à quoi elle sert, quels paramètres elle prend et quelle(s) valeur(s) elle retourne.

  tp3.c : contient le corps des fonctions déclarées dans tp3.h. Il doit inclure l'instruction *#include "tp3.h"*.

  tp3_main.c : contient uniquement la fonction *main()*. L'instruction *#include "tp3.h"* doit y être aussi.