Documentation développeur

Bruce MILPIED, Bowen ZHANG, Nicolas SUDRES, Ronan KOMPF

Introduction

Ce projet est réalisé dans le cadre de la matière `Temps Réel`. Il a pour objectif d'utiliser le langage C pour réaliser des calculs liés à la gestion de tâches en temps réel.

Dans une première partie, nous allons générer des chronogrammes pour un système de tâches périodiques. Pour cela, nous réalisons un programme prenant en arguments un système de tâches, une durée d'exécution et un nom d'algorithme.
Les algorithmes disponibles sont les suivants :

- FP (fixed priorities) : les priorités sont définies par l'ordre donné dans le fichier.
- EDF (dynamic priorities) : les priorités sont données par les échéances absolues des tâches.

Un programme réalisé en Python permet de générer une version graphique du résultat du programme C.

Dans une seconde partie, nous allons calculer le pire temps de réponse pour une tâche donnée. Pour ce programme, seul l'algorithme FP est utilisé. Le programme ne prend en argument que le fichier contenant le système de tâches.

Sommaire

Part 1

Architecture

Le programme est organisé de la manière suivante :

```
|-- part1/
|   |-- lib/
|   |   |-- algo.c
|   |   |-- algo.h
|   |   |-- part1.c
|   |   |-- part1.h
|   |   |-- sorted_job_list.c
|   |   `-- sorted_job_list.h
|   |-- graph.py
|   `-- makefile
```

La fonction `main` de notre programme est dans le fichier `part1.c`, c'est ici que nous traitons les arguments :

- Durée d'exécution
- Fichier
- Algorithme

Pour le fichier, nous récupérons le nombre de tâches, puis on enregistre nos tâches dans une structure `task`.
On choisit ensuite l'algorithme en fonction de l'argument.
Les deux algorithmes sont dans le fichier `algo.c`. Ils permettent tous les deux de générer sur une durée définie la tâche que le processeur doit exécuter à un moment `t`.
On affiche ensuite le résultat dans le terminal et on crée un fichier `output` avec ce même résultat.

Le fichier makefile permet de compiler les dépendances utilisées par `part1.c` dans le dossier `lib/` (`algo.c` et `sorted_job_list.c`), puis de créer l'exécutable `part1`.

Le fichier `graph.py` permet quant à lui de générer une visualisation graphique du résultat contenu dans le fichier `output`.

Fonctions

`void print(int result[], int F)` **(ligne 3 - part1.c)**

Arguments

- int result : tableau contenant les résultats
- int F : taille du tableau (correspond au nombre d'unités de temps)

Fonctionnement

Crée un fichier `output` contenant le résultat du programme. Ce fichier est utilisé pour l'affichage graphique (`graph.py`).

`void output(int result[], int F)` **(ligne 12 - part1.c)**

Arguments

- int result : tableau contenant les résultats
- int F : taille du tableau

Fonctionnement

Permet d'afficher les résultats du programme dans le terminal.

`int * fp(task tache[], int F, int n)` **(ligne 3 - algo.c)**

Arguments

- task tache : tableau d'une structure `task`, il contient Cn, Dn, Tn pour chaque tâche
- int F : taille du tableau (correspond au nombre d'unités de temps)
- int n : correspond au nombre de tâches

Fonctionnement

La fonction est composée d'une boucle for allant de 0 à F. A chaque passage, deux choses sont réalisées :

On vérifie si une instance d'une tâche doit être ajoutée dans notre tableau de tâches. Ce tableau contient le nombre d'unités de temps qu'il reste pour effectuer une tâche. Ex :

Tâche 1	Tâche 2	Tâche 3
2	5	3

Pour vérifier si on doit ajouter une nouvelle instance, on vérifie si la formule suivante est égale à t :
`(t/tache[i].Tn)*tache[i].Tn == t`
Si c'est le cas, il faut ajouter dans la case de la tâche i : `tache[i].Cn.`

Dans un second temps, on détermine pour le temps t, quelle tâche est exécutée. Pour cela, on enlève 1 à la tâche la plus prioritaire dans le tableau ci-dessus.

Return

Le return est un tableau de int contenant le résultat de l'algorithme. Le tableau doit être un tableau de taille `F`.

`int * edf(task tache[], int F, int K)` **(ligne 35 - algo.c)**

Arguments

- task tache : tableau d'une structure `task`, il contient Cn, Dn, Tn pour chaque tâche
- int F : taille du tableau (correspond au nombre d'unités de temps)
- int K : correspond au nombre de tâches

Fonctionnement

On crée dans un premier temps une liste vide avec la fonction `create_empty_list()`. On ajoute ensuite les différentes tâches avec la fonction `add_job()` dans une boucle for qui s'exécute K fois.
On fait ensuite une boucle for qui s'exécute F fois :

- Une boucle for vérifie s'il faut ajouter de nouveau les tâches : si `(t/tache[i].Tn)*tache[i].Tn) == t` alors on ajoute de nouveau la tâche en question avec `add_job()`.
- La case du tableau de résultats est calculée et remplie avec la fonction `schedule_first()`.

Return

Le return est un tableau de int contenant le résultat de l'algorithme. Le tableau doit être un tableau de taille `F`.

`void add_job(SortedJobList * job_list, int i, int c, int d)` **(ligne 7 - sorted_job_list.c)**

Arguments

- SortedJobList * job_list : liste dans laquelle on veut ajouter un élément
- int i : numéro de la tâche à ajouter
- int c : durée d'exécution pire cas
- int d : échéance relative (Dn+t)

Fonctionnement

On crée un nouvel élément dans lequel on stocke le `i, c, d`. Puis, selon le numéro de la tâche, on l'ajoute dans la liste. Si l'id est plus petit que le premier élément de la liste, on l'ajoute au début. Si c'est le plus grand, on ajoute à la fin de la liste. Sinon, on l'ajoute dans la liste entre des éléments plus petit/plus grand.

`SortedJobList create_empty_list()` **(ligne 45 - sorted_job_list.c)**

Return

La fonction ne contient qu'un return, ce return est une variable de type SortedJobList égale à `NULL`.

`void free_list(SortedJobList* job_list)` **(ligne 49 - sorted_job_list.c)**

Arguments

- SortedJobList* job_list : liste que l'on veut supprimer

Fonctionnement

Cette fonction a pour objectif de libérer les espaces mémoire utilisés par une liste. Pour cela, il y a une boucle for qui libère la liste case par case.

`int schedule_first(SortedJobList * job_list)` **(ligne 59 - sorted_job_list.c)**

Arguments

- SortedJobList * job_list :

Fonctionnement

Cette fonction vérifie dans un premier temps si le job n'est pas vide. Si c'est le cas, elle fait -1 puis vérifie si le job est à 0. Si c'est le cas, la fonction libère la mémoire et redéfinit le job suivant en tant que premier.

Return

Retourne le numéro de tâche à exécuter en priorité.

Bug report

Pas de bug

Part 2

Architecture

Le programme est organisé de la manière suivante :

```
|-- part2/
|   |-- lib/
|   |   |-- worst_case_fp.c
|   |   |-- worst_case_fp.h
|   |   |-- part2.c
|   |   `-- part2.h
|   `-- makefile
```

La fonction main de notre programme est dans le fichier `part2.c`, c'est ici que nous traitons l'argument fichier.
Pour le fichier, nous récupérons le nombre de tâches, puis on enregistre nos tâches dans une structure `task`.

On appelle ensuite la fonction `compute()`.
Cette fonction appelle ensuite une succession de fonctions permettant le calcul du pire temps de réponse : `get_worst_case_responce_time()`, `get_nb_critical_job()`, `get_busy_period()`, `get_responce_time`.

Le fichier makefile permet de compiler les dépendances utilisées par `part2.c` dans le dossier `lib/` (`worst_case_fp.c`), puis de créer l'exécutable `part2`.

Fonctions

`void compute(Taskset tache[], int nb_tache)` **(ligne 3 - part2.c)**

Arguments

- Taskset tache[] : tableau d'une structure taskset, il contient Cn, Dn, Tn pour chaque tâche
- int nb_tache : nombre de tâches dans la structure `Taskset`

Fonctionnement

Permet de faire l'affichage des return en fonction du nombre de tâches. L'affichage est au format suivant :

```
Tâche X :
Résultat de la fonction get_busy_period : X
Nombre d'instances pendant la busy period  : X
Pire temps de réponse : X
```

Pour cela, `compute()` appelle la fonction `get_worst_case_responce_time`.

`int test_load(Taskset tache[], int nb_tache)` **(ligne 3 - worst_case_fp.c)**

Arguments

- Taskset tache[] : tableau d'une structure taskset, il contient Cn, Dn, Tn pour chaque tâche
- int nb_tache : nombre de tâches dans la structure `Taskset`

Fonctionnement

Pour calculer les charges d'un ensemble de tâches, la fonction utilise la structure pseudo code suivante :

```
borne = nb_tache*(pow(2,1.0/nb_tache)-1)

Pour toutes les tâches :
    Si Cn <= Tn
        Si Cn == Tn
            load = somme(Cn/Tn)
        Sinon
            On vérifie que les tâches ont des échéances croissantes
            load = somme(Cn/Dn)
        Fin si
    Sinon
        Erreur D n'est pas inférieure à T
    Fin si

Si load <= borne
    return 1
Sinon si result <= 1
    return 0
Sinon
    return -1
Fin si
```

Return

La fonction retourne 1 si la charge est inférieure à la borne, 0 si la charge est entre la borne et 1, -1 si la charge est supérieure à 1.

`int get_busy_period(Taskset tache[], int i)` **(ligne 50 - worst_case_fp.c)**

Arguments

- Taskset tache[] : tableau d'une structure taskset, il contient Cn, Dn, Tn pour chaque tâche
- int i : numéro de la tâche utilisée dans la fonction

Fonctionnement

La fonction est composée d'une boucle Do While. Tant que t (qui s'incrémente à chaque passage) n'est pas égal à la busy_period (busy_period=(T/tache[k].Tn)*tache[k].Cn), la boucle continue. k étant un entier entre 0 et i. Cela permet de prendre en compte les tâches prioritaires par rapport à celle utilisée pour le calcul.

Return

La fonction retourne la busy period de la tâche passée en argument.

`int get_nb_critical_job(Taskset tache[], int i, int bp)` **(ligne 66 - worst_case_fp.c)**

Arguments

- Taskset tache[] : tableau d'une structure taskset, il contient Cn, Dn, Tn pour chaque tâche
- int i : numéro de la tâche utilisée dans la fonction
- int bp : busy période d'une tâche i

Fonctionnement

Cette fonction est une simple division de bp/tache[i].Tn. Cela permet de déterminer le nombre d'instances de notre tâche durant la `busy_period`.

Return

La fonction retourne le nombre d'instances de la tâche passée en argument et la busy period calculée précédemment.

`int get_responce_time(Taskset tache[], int i, int k)` **(ligne 74 - worst_case_fp.c)**

Arguments

- Taskset tache[] : tableau d'une structure taskset, il contient Cn, Dn, Tn pour chaque tâche
- int i : numéro de la tâche utilisée dans la fonction
- int k : numéro de l'instance pour laquelle on veut calculer le temps de réponse

Fonctionnement

Cette fonction calcule le temps de réponse d'une tâche. Pour cela, elle fait le calcul suivant : responce_time = date_terminaison - date_activation

date_terminaison : boucle while qui s'arrête quand t == date_terminaison. t prend la valeur précédente de date_terminaison et date_terminaison prend la valeur (t/tache[i].Tn)*tache[i].Cn

date_activation : (k-1)*tache[i].Tn

Return

La fonction retourne le temps de réponse d'une tâche en fonction de son numéro d'instance.

`int get_worst_case_responce_time(Taskset tache[], int i)` **(ligne 90 - worst_case_fp.c)**

Arguments

- Taskset tache[] : tableau d'une structure taskset, il contient Cn, Dn, Tn pour chaque tâche
- int i : numéro de la tâche utilisée dans la fonction

Fonctionnement

Cette fonction prend la valeur maximale du temps de réponse d'une tâche i. Pour cela, elle calcule dans un premier temps le nombre de jobs critiques avec la fonction `get_nb_critical_job()`. Puis, elle calcule le temps de réponse de chaque instance de la tâche avec la fonction `get_responce_time()`. Elle affiche au final le pire temps de réponse parmi les différentes instances.

Return

La fonction retourne le pire temps de réponse pour un ensemble d'instances d'une même tâche.

Bug report

Pas de bug

Conclusion

//TODO