FOLLEREAU Juliane

5TS1

IPSA
ÉCOLE D'INGÉNIEURS
DE L'AIR, DE L'ESPACE ET DE LA MOBILITÉ DURABLE

Real-time operating system for embedded systems

FINAL ASSIGNMENT

20/01/2025

Exercice 1 : planification

Explication du code :

- La fonction *ppcm(a,b)* : cette fonction calcule le plus petit multiple commun entre deux nombres a et b, on l'utilise pour déterminer l'hyper-période des tâches.
- Les fonctions *calculer_temps_de_reponse* et *calculer_temps_de_reponse2* : ces deux fonctions font le calcul des temps des jobs selon l'ordonnancement. Pour l'ordonnancement j'ai tester avec deux approches différentes
    - Dans *calculer_temps_de_reponse* les jobs sont exécutés selon un ordre où le premier job arrivé est exécuté en premier. On suit l'ordre d'arrivée des jobs.
    - Dans *calculer_temps_de_reponse2* les jobs sont exécutés selon l'ordre où le job avec la deadline la plus proche est choisi en priorité. On respect ainsi mieux les deadlines et on peut être mieux réduire les violations de deadlines…

  J'ai voulu tester avec plusieurs approches de sélection des jobs afin de voir si déjà ça avait un impact sur la planification et ensuite si ça allait changer le temps d'attente.

La logique de planification :

- **Création des jobs** : chaque tâche est représentée par un ensemble de jobs créés en fonction de la période de la tâche (répétition tous les T_i cycles). Chaque job a un nom, un temps d'exécution (C_i), un temps de période (T_i), un temps d'arrivée et un temps restant.
- **Simulation de l'exécution des jobs** : A chaque cycle de temps (valeur de *temps_courant*), les jobs prêts à être exécutés (arrivés avant ou à *temps_courant* et non terminés) sont choisis. L'exécution est simulée en réduisant leur temps restant et, lorsque terminé, le temps de réponse est calculé.
- **Vérification des deadlines** : Après chaque exécution, le code vérifie si un job a manqué sa deadline. Si c'est le cas, la tâche est ajoutée à la liste des tâches manquées.

Gestion des ordonnancements : Le code génère toutes les permutations possibles des ordonnancements de tâches, puis simule l'exécution avec la fonction correspondante. Le meilleur ordonnancement minimise le temps d'attente total tout en respectant les deadlines ou en autorisant une violation de la deadline pour la tâche 5.

Temps de calcul : tâches vs jobs : Travailler directement avec les tâches plutôt qu'avec les jobs a permis de réduire le temps de calcul (considérablement !!). En effet, lorsqu'on travail avec les jobs, chaque permutations de jobs implique un calcul pour toutes les instances possibles, ce qui augmente considérablement la complexité. En utilisant les tâches, le nombre de permutations à tester est réduit à n ! où n est le nombre de tâches, ce qui rend l'algorithme beaucoup plus rapide et adapté pour trouver la meilleure planification sous un coût de calcul trop élevé.

Calcul du temps d'inactivité du processeur : Le temps d'inactivité du processeur est calculé chaque fois qu'aucun job n'est prêt à être exécuté. Ce cas se produit lorsque les jobs ont tous un temps d'arrivée supérieur au temps courant ou lorsque les jobs en cours d'exécution sont terminés. Chaque cycle où le processeur reste inactif est comptabilisé, et à la fin de l'exécution de tous les ordonnancements, le temps total d'inactivité est affiché. En utilisant chacune de nos deux fonctions ce temps est de 0 et de 15, cela signifie que les ordonnancements sont efficaces pour l'un des cas mais que pour le second le processeur est sous-utilisé…

Comparaison des résultats :

Pour ces résultats j'ai utilisé la première consigne que vous aviez fourni ! en utilisant la seconde consigne avec le 3 dans la tâche 2 la planification n'étais pas schedulable avec la fonction *calculer_temps_de_reponse* … Le résultat était le suivant :

```
1. Aucun ordonnancement valide sans deadline manquée.
2. Tâches ayant manqué leur deadline : T2
3.
4. Aucun ordonnancement valide même en autorisant T5 à manquer une deadline.
5. Tâches ayant manqué leur deadline : T2
6.
```

J'ai donc utiliser la consigne précédente qui était :

```
 1. # définition des tâches : (nom, C_i, T_i)
 2. taches = [
 3.     ("T1", 2, 10),
 4.     ("T2", 2, 10),
 5.     ("T3", 2, 20),
 6.     ("T4", 2, 20),
 7.     ("T5", 2, 40),
 8.     ("T6", 2, 40),
 9.     ("T7", 3, 80)
10. ]
11.
```

- **Temps d'exécution total** : le temps total est un peu plus court quand on utilise la fonction *calculer_temps_de_reponse2* (job avec la deadline la plus proche exécutée en premier) et vaut 175 pour la meilleure permutation. Quand on utilise la fonction *calculer_temps_de_reponse* (premier job arrivé = premier exécuté) le temps passe à 177 pour la meilleure permutation.

- **Planification** : Une fois de plus selon la fonction utilisée la meilleure planification est différente puisque les jobs sont exécutés selon leur deadline ou leur ordre d'arrivée.

    - Avec *calculer_temps_de_reponse2* :

```
 1. Meilleur ordonnancement sans deadline manquée :
 2.   Job T1 - Arrivée : 0, Fin : 2, Temps de réponse : 2
 3.   Job T2 - Arrivée : 0, Fin : 4, Temps de réponse : 4
 4.   Job T3 - Arrivée : 0, Fin : 6, Temps de réponse : 6
 5.   Job T4 - Arrivée : 0, Fin : 8, Temps de réponse : 8
 6.   Job T5 - Arrivée : 0, Fin : 10, Temps de réponse : 10
 7.   Job T1 - Arrivée : 10, Fin : 12, Temps de réponse : 2
 8.   Job T2 - Arrivée : 10, Fin : 14, Temps de réponse : 4
 9.   Job T6 - Arrivée : 0, Fin : 16, Temps de réponse : 16
10.   Job T7 - Arrivée : 0, Fin : 19, Temps de réponse : 19
11.   Job T1 - Arrivée : 20, Fin : 22, Temps de réponse : 2
12.   Job T2 - Arrivée : 20, Fin : 24, Temps de réponse : 4
13.   Job T3 - Arrivée : 20, Fin : 26, Temps de réponse : 6
14.   Job T4 - Arrivée : 20, Fin : 28, Temps de réponse : 8
15.   Job T1 - Arrivée : 30, Fin : 32, Temps de réponse : 2
16.   Job T2 - Arrivée : 30, Fin : 34, Temps de réponse : 4
17.   Job T1 - Arrivée : 40, Fin : 42, Temps de réponse : 2
18.   Job T2 - Arrivée : 40, Fin : 44, Temps de réponse : 4
19.   Job T3 - Arrivée : 40, Fin : 46, Temps de réponse : 6
20.   Job T4 - Arrivée : 40, Fin : 48, Temps de réponse : 8
21.   Job T5 - Arrivée : 40, Fin : 50, Temps de réponse : 10
22.   Job T1 - Arrivée : 50, Fin : 52, Temps de réponse : 2
23.   Job T2 - Arrivée : 50, Fin : 54, Temps de réponse : 4
24.   Job T6 - Arrivée : 40, Fin : 56, Temps de réponse : 16
25.   Job T1 - Arrivée : 60, Fin : 62, Temps de réponse : 2
26.   Job T2 - Arrivée : 60, Fin : 64, Temps de réponse : 4
```

```
27.   Job T3 - Arrivée : 60, Fin : 66, Temps de réponse : 6
28.   Job T4 - Arrivée : 60, Fin : 68, Temps de réponse : 8
29.   Job T1 - Arrivée : 70, Fin : 72, Temps de réponse : 2
30.   Job T2 - Arrivée : 70, Fin : 74, Temps de réponse : 4
31. Temps total d'attente : 175
32. Temps d'inactivité : 15
```

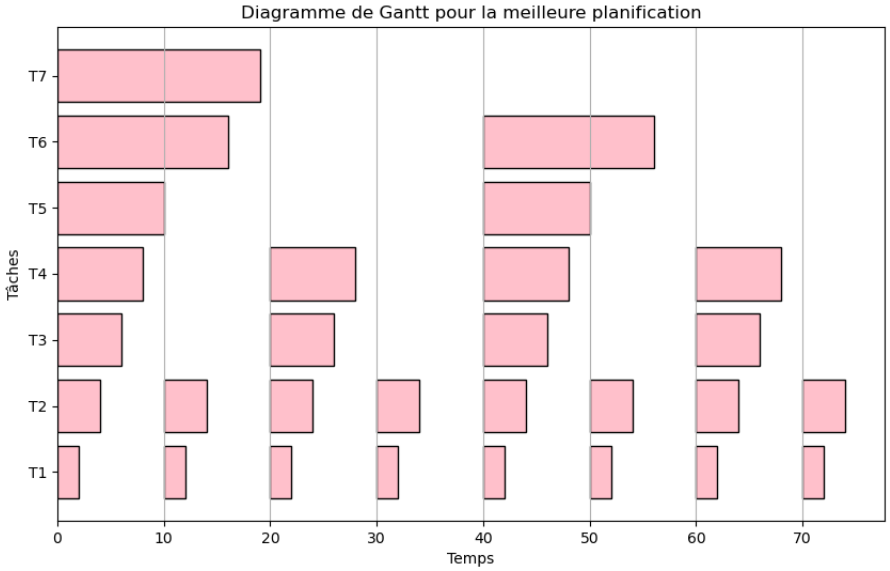
- Avec *calculer_temps_de_reponse* :

```
 1. Meilleur ordonnancement sans deadline manquée :
 2.   Job T1 - Arrivée : 0, Fin : 2, Temps de réponse : 2
 3.   Job T2 - Arrivée : 0, Fin : 4, Temps de réponse : 4
 4.   Job T3 - Arrivée : 0, Fin : 6, Temps de réponse : 6
 5.   Job T4 - Arrivée : 0, Fin : 8, Temps de réponse : 8
 6.   Job T5 - Arrivée : 0, Fin : 10, Temps de réponse : 10
 7.   Job T6 - Arrivée : 0, Fin : 12, Temps de réponse : 12
 8.   Job T7 - Arrivée : 0, Fin : 15, Temps de réponse : 15
 9.   Job T1 - Arrivée : 10, Fin : 17, Temps de réponse : 7
10.   Job T2 - Arrivée : 10, Fin : 19, Temps de réponse : 9
11.   Job T1 - Arrivée : 20, Fin : 22, Temps de réponse : 2
12.   Job T2 - Arrivée : 20, Fin : 24, Temps de réponse : 4
13.   Job T3 - Arrivée : 20, Fin : 26, Temps de réponse : 6
14.   Job T4 - Arrivée : 20, Fin : 28, Temps de réponse : 8
15.   Job T1 - Arrivée : 30, Fin : 32, Temps de réponse : 2
16.   Job T2 - Arrivée : 30, Fin : 34, Temps de réponse : 4
17.   Job T1 - Arrivée : 40, Fin : 42, Temps de réponse : 2
18.   Job T2 - Arrivée : 40, Fin : 44, Temps de réponse : 4
19.   Job T3 - Arrivée : 40, Fin : 46, Temps de réponse : 6
20.   Job T4 - Arrivée : 40, Fin : 48, Temps de réponse : 8
21.   Job T5 - Arrivée : 40, Fin : 50, Temps de réponse : 10
22.   Job T6 - Arrivée : 40, Fin : 52, Temps de réponse : 12
23.   Job T1 - Arrivée : 50, Fin : 54, Temps de réponse : 4
24.   Job T2 - Arrivée : 50, Fin : 56, Temps de réponse : 6
25.   Job T1 - Arrivée : 60, Fin : 62, Temps de réponse : 2
26.   Job T2 - Arrivée : 60, Fin : 64, Temps de réponse : 4
27.   Job T3 - Arrivée : 60, Fin : 66, Temps de réponse : 6
28.   Job T4 - Arrivée : 60, Fin : 68, Temps de réponse : 8
29.   Job T1 - Arrivée : 70, Fin : 72, Temps de réponse : 2
30.   Job T2 - Arrivée : 70, Fin : 74, Temps de réponse : 4
31. Temps total d'attente : 177
32. Temps d'inactivité : 0
```

- **Deadline manquée** : Pour les deux fonctions, dans le cas où la tâche 5 est autorisé à manquer sa deadline, les meilleures planifications sont identiques à celles-ci-dessus.
- **Temps d'inactivité** : Le temps d'inactivité varié selon la fonction que nous utilisons en passant de 0 à 15. Comme quoi minimiser le temps d'exécution n'est pas synonyme d'une inactivité du processeur la meilleure…

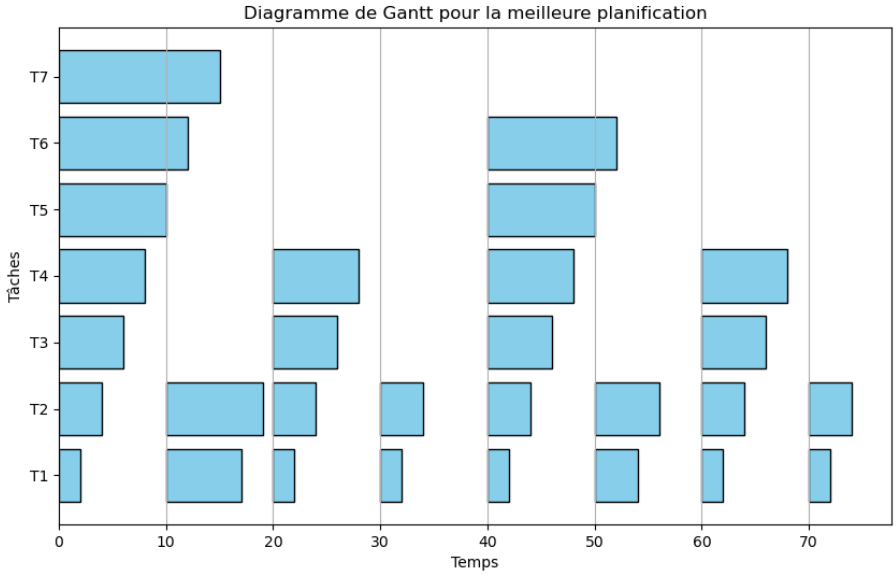
Illustration : J'ai décidé de représenter les meilleur ordonnancement avec chacune des deux fonctions que j'ai utilisé afin de rendre les choses un peu plus visuel.

Avec *calculer_temps_de_reponse2* :



Cette figure montre l'ordonnancement quand les jobs sont exécutés en fonction de leur deadline, avec une priorité donnée à ceux dont la deadline est la plus proche. Cette approche réduit les risques de violations de deadlines.

Avec *calculer_temps_de_reponse* :



Cette figure montre l'ordonnancement des jobs dans l'ordre d'arrivée, indépendamment de leurs deadlines. Elle met en évidence comment un job avec une deadline éloignée peut être exécuté avant un job à échéance plus proche, ce qui peut entraîner des violations de deadlines.

Conclusion : Nous avons observé que l'approche avec la fonction *calculer_temps_de_reponse2* réduit le temps total à 175, contre 177 dans le cas de la fonction *calculer_temps_de_reponse* . Cependant, bien que cette approche semble optimiser le respect des deadlines, elle entraine un certain temps d'inactivité du processeur de 15 unités, ce qui suggère une sous-utilisation des ressources à certains moments. Cela met en évidence qu'une planification visant à minimiser le temps d'exécution ne garantit pas nécessairement une utilisation optimale du processeur.

FreeRTOS

Concernant cet exercice j'ai repris mon code de l'année dernière qui fonctionnait déjà au niveau de toutes les tâches ! j'ai ajouté la tâche 5, voici une explication détaillé :

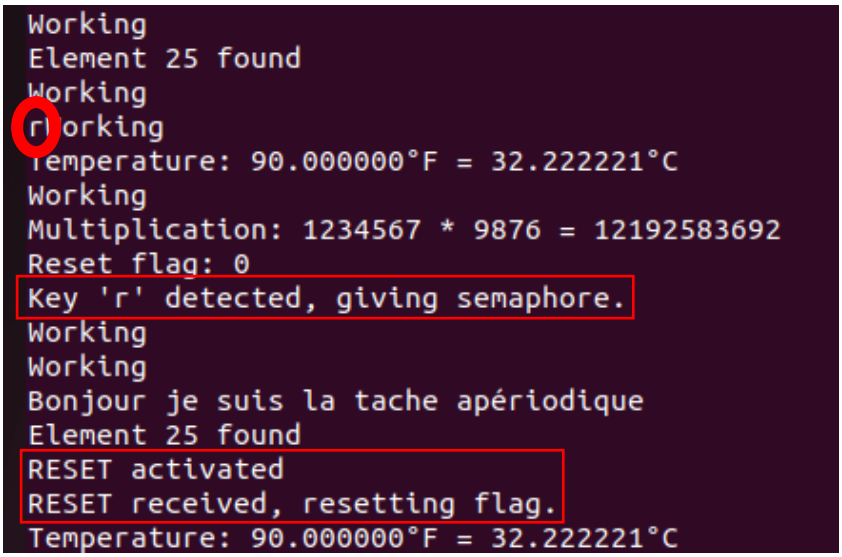
Fonction *kbhit()* pour la vérification de la pression de la touche 'r': Cette fonction permet de vérifier si une touche (ici la touche 'r') a été pressée sans bloquer l'exécution du programme. Elle commence par sauvegarder les paramètres du terminal, puis elle les modifie pour rendre la lecture de l'entrée clavier non-bloquante. Le terminal passe en mode non-canonique, ce qui permet de lire les caractères sans attendre que l'utilisateur appuie sur 'entrée'. Ensuite, elle utilise *getchar()* pour détecter une touche pressée. Si une touche est pressée, la fonction remet le caractère dans le buffer d'entrée et retourne 1. Si aucune touche n'a été pressée, elle retrouve 0. Cette fonction est utilisé pour vérifier en temps réel l'entrée utilisateur, sans interrompre le programme en cours d'exécution.

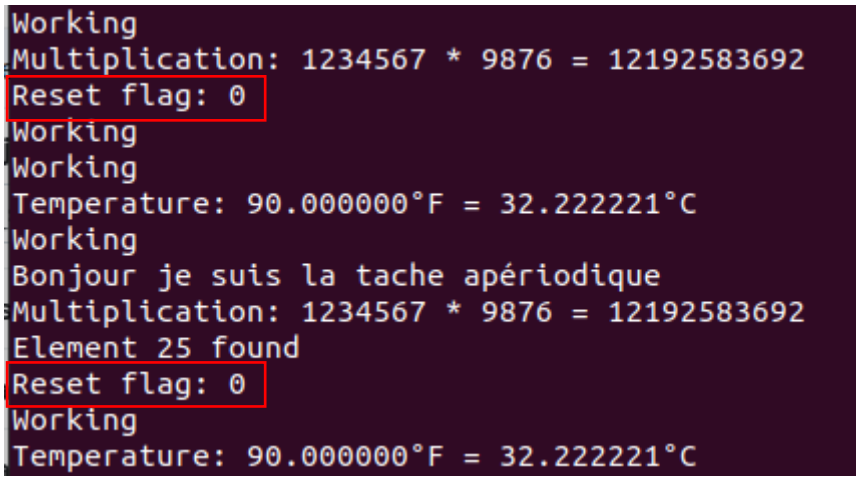
Fonction *simulateButtonPres()* pour la simulation de pression de touche : Cette fonction s'appuie sur la fonction précédente *kbhit()* pour détecter si une touche a été pressée. Si une touche est détectée, elle est lue avec *getchar()*. Si la touche pressée est la lettre 'r', le fonction libère le sémaphore *xButtonSemaphore*, signalant ainsi à la tâche 5 qu'un RESET est demandé. Cette fonction permet d'implémenter une action concrète en réponse à la pression d'une touche spécifique ici la touche 'r', pour déclencher un processus dans le système tel qu'une réinitialisation.

Objectifs de la tâche 5 : La tâche 5 (prvPeriodicTask5) est responsable de régir à l'évènement de la pression de la touche 'r' en activant un RESET du système. Voici une explication de son fonctionnement :

- **Initialisation du flag** : La tâche commence par initialiser un flag *reset_flag* à zéro, ce qui signifie qu'aucune demande de RESET n'a été effectuée par défaut.
- **Boucle infinie** : La tâche entre dans une boucle infinie qui lui permet de surveiller l'état du sémaphore et de détecter toute demande de RESET.
- **Attente du sémaphore** : A chaque itération de la boucle, la tâche attend un signal du sémaphore via *xSemaphoreTake(xButtonSemaphore*, *pdMS_TO_TICKS(200))*. Si le sémaphore est libéré par la fonction *simulateButtonPres()* ça indique qu'un RESET a été demandé.
- **Activation du RESET** : Lorsque le sémaphore est libéré, la tâche réagit en activant le RESET, ce qui est effectué en mettant à jour le flag *reset_flag* et en affichant un message indiquant que le RESET a été activé. Cette étape garantit que le système réagit à la demande de la réinitialisation.
- **Affichage et réinitialisation du flag** : Après activation du RESET, la tâche affiche un message et remet le *reset_flag* à zéro. Si aucune autre demande de RESET n'a été détectée, elle affiche juste l'état actuel du flag.
- **Détection continue de la touche** : La tâche appelle la fonction *simulateButtonPres()* pour vérifier en permanence si l'utilisateur appuie sur la touche 'r', permettant ainsi une détection en temps réel de l'entrée extérieur de l'utilisateur.
- **Pause avant la prochaine vérification** : Avant de redémarrer la boucle, la tâche attend 200ms via *vTaskDelay(TASK5_PERIOD_MS)*.

Voici ce que nous pouvons observer si on presse la touche 'r' lors de l'exécution du programme :





Rappel sur le fonctionnement des autres tâches :

- ✓ Tâche 1 : Cette tâche s'exécute toute les 100 ms et affiche le message « Working » à chaque itération. Son objectif principal est de simuler une tâche qui fonctionne en permanence pour montrer l'activité d'un processus récurrent.
- ✓ Tâche 2 : Cette tâche est responsable de la conversion d'une température donnée de Fahrenheit à Celsius. Elle s'exécute toutes les 300 ms, calcule la conversion, puis affiche les résultats.
- ✓ Tâche 3 : Cette tâche effectue une multiplication entre deux entiers et affiche le résultat toute les 400ms. Le but est de réaliser une opération arithmétique simple et de la présenter sous forme de sortie dans la console.
- ✓ Tâche 4 : Cette tâche effectue une recherche binaire pour trouver un élément dans une liste triée de 50 entiers. Elle s'exécute toutes les 500 ms et affiche si l'élément recherché est trouvé ou non.

- ✓ Fonctionnement global : Le code implémente un système de gestion de tâches en utilisant FreeRTOS. Ce système crée plusieurs tâches périodiques (et une apériodique de l'année dernière). Les tâches sont planifiées pour être exécutées à des intervalles spécifiques. Le programme initialise des sémaphores, des files d'attente et gère l'interaction avec l'utilisateur, notamment avec la détection de le pression de la touche 'r' !! Les tâches s'exécutent simultanément en fonction de leur priorité, garantissant une gestion efficace des ressources.