Candice Giami & Clément Szewczyk

Rapport du Projet d'Architecture Système

Le Gestionnaire de tâches différées

Introduction :

À travers ce rapport, nous documentons en détail notre approche pour répondre à ce défi, mettant l'accent sur la modularité, la simplicité d'utilisation et la flexibilité. Nous explorerons les choix de conception effectués pour permettre une gestion efficace des tâches différées, en fournissant une analyse approfondie du problème et une solution pratique pour une planification dynamique.

En respectant les contraintes du langage C standard, nous avons développé des structures de données efficaces, des fonctions modulaires et des mécanismes de gestion des tâches, offrant ainsi une expérience utilisateur fluide et intuitive. Ce rapport servira de guide complet, fournissant une documentation détaillée de notre code, un mode d'emploi approfondi.

Rejoignez-nous dans cette aventure de développement, où chaque ligne de code contribue à façonner un environnement propice à des tâches différées efficaces et à des processus optimisés.

Objectif du Projet :

L'objectif principal de notre projet est de concevoir et de mettre en œuvre un gestionnaire de tâches différées en langage C. Cette application fournira aux utilisateurs un moyen simple et efficace de planifier et d'exécuter des tâches à des moments précis. L'interface utilisateur intuitive permettra aux utilisateurs de créer, visualiser et gérer facilement leurs tâches différées.

Détails :

Contraintes :

- Nous habitons très loin l'un de l'autre, nous avons donc dû nous organiser à distance via Git Hub.
- Nous étions ainsi obligées de se partager le travail plutôt que de travailler tous les 2 ensembles simultanément.

L'équipe :

L'équipe pour réaliser ce projet est composé de Candice et de Clément.

Mode de travail :

Nous avons entamé la réalisation du projet peu après les vacances de février, et avons rapidement achevé la version minimale. Pour répartir efficacement le travail, nous avons adopté une approche collaborative :

- Clément a élaboré la première version du code en utilisant des fonctions de certaines bibliothèques interdites, dont nous discuterons plus en détail ultérieurement dans la chronologie du projet. Par la suite, Candice a pris la relève et a refait tout le projet. Clément s'est occuper de rajouter au code une fonction pour reprendre la main sur le terminal durant une exécution.

- Candice s'est concentrée sur la rédaction du rapport.

Nous avons maintenu une communication régulière sur Discord pour partager nos problèmes et nous apporter mutuellement de l'aide. Nous avons convenu ensemble des tâches à effectuer à l'avance afin d'assurer une cohésion harmonieuse de notre travail, malgré nos emplois du temps respectifs qui ne nous permettaient pas de nous voir fréquemment. Nous avons également veillé à mettre à jour le code sur GitHub régulièrement pour faciliter la collaboration.

Présentation du plan :

Table des matières

Développement :

La réalisation :

Ce que nous devons faire :

Dans le cadre de notre projet, nous développons un logiciel de gestion de tâches différées en langage C. Ce logiciel offre une solution flexible pour la gestion des tâches informatiques en permettant l'exécution différée de commandes. Notre objectif principal consiste à développer une version minimale du programme qui permettra la planification des exécutions itérées d'une commande. L'utilisateur pourra dynamiquement spécifier les paramètres suivants lors de l'exécution du planificateur :

- La commande à exécuter ;

- Le délai entre chaque exécution ;

- Le nombre d'itérations de la commande.

Il est important de noter que ces paramètres ne sont pas prédéfinis lors de la compilation, mais sont fournis au programme lors de son exécution, offrant ainsi une grande flexibilité.

Dans une version plus avancée du programme, nous prévoyons d'ajouter des fonctionnalités supplémentaires pour enrichir l'expérience utilisateur. Parmi ces fonctionnalités, nous envisageons la possibilité de programmer une date précise pour l'exécution des tâches avant de lancer le planificateur. De plus, nous pourrions implémenter la capacité pour le planificateur de gérer un nombre infini d'itérations pour une plus grande adaptabilité aux besoins des utilisateurs.

La structure du code :

La version minimale de notre projet comprend quatre fonctions principales :

La fonction « main » est le point d'entrée du programme. Voici ce qu'elle fait :

1. Elle vérifie le nombre d'arguments passés à partir de la ligne de commande. Si le nombre d'arguments n'est pas égal à 4 ou 5, elle affiche un message d'utilisation indiquant la manière correcte d'utiliser le programme et quitte avec un code de retour d'erreur.

2. Elle analyse les arguments passés en ligne de commande pour récupérer le nombre d'itérations, le délai, la commande à exécuter et éventuellement une date spécifique.

3. Elle vérifie la validité du délai et du nombre d'itérations, affichant des messages d'erreur appropriés si nécessaire.

4. Si une date est spécifiée, elle la compare avec la date actuelle pour déterminer si elle est antérieure, identique ou postérieure. En fonction de cela, elle exécute immédiatement la commande ou attend jusqu'à la date spécifiée avant de lancer la planification des tâches.

5. Si aucune date n'est spécifiée, elle lance immédiatement la planification des tâches en appelant la fonction « fork_planifier » avec les paramètres appropriés.

6. Le programme se termine en retournant 0 pour indiquer une exécution réussie.

La fonction « fork_planifier » crée un processus fils pour exécuter la planification des tâches. Voici ce qu'elle fait en détail :

Elle prend en paramètres la commande à exécuter, le délai entre chaque itération et le nombre d'itérations.

Elle utilise la fonction fork() pour créer un nouveau processus fils.

Dans le processus fils, elle appelle la fonction « planifier_taches » avec les paramètres fournis, ce qui lance l'exécution de la commande avec la planification appropriée.

Le processus fils se termine une fois que la fonction « planifier_taches » a terminé son exécution.

Le processus parent continue son exécution normale après avoir créé le processus fils.

La fonction « planifier_taches » est responsable de la planification des tâches en fonction du délai et du nombre d'itérations. Voici ce qu'elle fait :

1. Elle prend en paramètres la commande à exécuter, le délai entre chaque itération et le nombre d'itérations.

2. Si le nombre d'itérations est infini (représenté par -1), elle entre dans une boucle infinie où elle crée un nouveau processus fils à chaque itération pour exécuter la commande.

3. Si le nombre d'itérations est fini, elle entre dans une boucle où elle crée un processus fils pour chaque itération, en attendant le délai entre chaque itération sauf pour la dernière.

4. Chaque processus fils exécute la commande en appelant la fonction « executer_commande ».

5. Une fois toutes les itérations terminées, le processus parent attend la fin des processus fils déjà lancés avant de se terminer.

La fonction « executer_commande » est chargée d'exécuter une commande via le shell. Voici ce qu'elle fait :

1. Prend en paramètre une chaîne de caractères représentant la commande à exécuter.

2. Affiche un message pour indiquer que la commande est en cours d'exécution.

3. Utilise la fonction « execl » pour exécuter la commande via le shell. Cette fonction remplace le processus courant par celui spécifié dans les arguments, en l'occurrence le shell `/bin/sh`, avec l'option -c pour lui passer la commande à exécuter.

4. Si « execl » échoue, affiche un message d'erreur avec « perror » et termine le processus en cours avec « exit(EXIT_FAILURE) ».

Dans la version finale, aucune nouvelle fonction n'a été ajoutée. Nous avons plutôt choisi d'enrichir les fonctionnalités existantes en ajoutant deux options supplémentaires :

1. L'option "infini" : Permettant d'exécuter un nombre infini d'itérations.
2. L'option de datation : Qui permet de démarrer le planificateur à une date fixée préalablement.

Comme leur nom l'indique, ces options sont facultatives et leur activation est simple.

Choix et aléas :

- ✓ Continuité des tâches :

  Choix : Nous avons opté pour une approche où l'utilisateur reprend la main entre chaque tâche planifiée.

  Avantages : Permettre à l'utilisateur de reprendre la main entre chaque tâche planifiée rend l'expérience utilisateur plus interactive. Cela évite que l'utilisateur se retrouve bloqué pendant les délais, surtout s'ils sont longs.

  Aléas : Toutefois, cette approche peut rendre la visualisation des résultats de chaque itération moins claire dans le terminal, car le flux de sortie peut être interrompu par les messages du système ou d'autres programmes. Cependant, il est toujours possible d'arrêter l'exécution du programme en utilisant la combinaison de touches CRTL+C si nécessaire.

- ✓ Choix de l'utilisateur directement dans le terminal :

  Avantages : Permet de maintenir la clarté du terminal en évitant les informations superflues qui pourraient réduire la lisibilité des résultats et la navigation dans le terminal.

  Aléas : Cette approche peut être moins intuitive pour un utilisateur novice en programmation. Pour remédier à cela, nous fournissons une explication du fonctionnement du programme lorsque l'utilisateur lance simplement le programme avec «./planificateur ».

- ✓ Gestion des Erreurs :

  Avantages : Nous avons inclus plusieurs messages de débogage pour aider l'utilisateur à résoudre les problèmes éventuels rencontrés lors de la planification des tâches.

  Aléas : Malgré nos efforts pour gérer les erreurs, des situations inattendues peuvent survenir lors de l'exécution du programme.

- ✓ Interprétation des arguments de ligne de commande :

  Choix : Le programme interprète les arguments de la ligne de commande pour déterminer le nombre d'itérations, le délai, la commande à exécuter et éventuellement la date spécifique.

  Avantages : Cette approche offre une flexibilité à l'utilisateur pour personnaliser la planification des tâches en fonction de ses besoins spécifiques.

Aléas : Une mauvaise utilisation des arguments en ligne de commande peut entraîner des erreurs d'exécution ou des comportements inattendus du programme.

- ✓ Gestion du temps :

  Choix : Le programme utilise les fonctions de gestion du temps pour attendre jusqu'à une date spécifiée avant de commencer l'exécution des tâches.

  Avantages : Cela permet de planifier des tâches à des moments précis, offrant ainsi une planification plus précise des opérations.

  Aléas : Les problèmes liés à la gestion du temps, tels que les fuseaux horaires incorrects ou les erreurs de conversion de date, peuvent entraîner des résultats imprévus ou une exécution incorrecte des tâches planifiées.

- ✓ Option infinie :

  Choix : Le programme permet à l'utilisateur de spécifier l'option "infini" pour exécuter la commande de manière répétée sans limite d'itérations.

  Avantages : Cette option offre une grande souplesse à l'utilisateur pour exécuter une commande de manière continue sans avoir à spécifier un nombre fixe d'itérations.

  Aléas : Cependant, l'exécution continue d'une commande peut entraîner une consommation excessive de ressources système, en particulier si la commande a un impact significatif sur les performances du système. Heureusement, dans le cas où l'utilisateur souhaite arrêter l'exécution en cours, il peut fermer le terminal.

- ✓ Spécification de la commande entre guillemets :

  Choix : Nous avons décidé d'exiger que l'utilisateur spécifie la commande entre guillemets afin que l'intégralité de la commande soit considérée comme un seul argument, même si elle contient des espaces.

Avantages : Cette approche garantit que la commande est interprétée comme un seul argument, évitant ainsi toute confusion ou erreur d'interprétation des espaces dans la commande.

Aléas : Bien que cela simplifie le traitement des arguments de la ligne de commande, certains utilisateurs peuvent oublier d'inclure les guillemets, ce qui peut entraîner des erreurs d'exécution du programme. Un rappel de cette exigence est fourni dans le message d'utilisation du programme.

- ✓ Unité de temps pour les délais :

  Choix : Nous avons choisi d'utiliser les secondes comme unité de temps pour les délais entre les itérations.

  Avantages : L'utilisation des secondes offre une granularité appropriée pour spécifier des délais courts ou longs, ce qui permet une flexibilité dans la planification des tâches.

  Aléas : Bien que les secondes soient couramment utilisées et facilement compréhensibles, cela peut poser un problème si une précision temporelle plus fine est requise, notamment pour des tâches nécessitant une minutie extrême dans la planification.

- ✓ Lancement immédiat si la date est antérieure :

  Choix : Nous avons décidé de permettre au programme de démarrer immédiatement l'exécution des tâches si la date spécifiée par l'utilisateur est antérieure à la date actuelle, plutôt que de générer une erreur.

  Avantages : Cela peut être utile dans certaines situations où l'utilisateur c'est trompé de date sans qu'il a besoin de rentrer a nouveaux toute les informations.

  Aléas : Bien que cela puisse être pratique dans certains cas, cela peut également entraîner des exécutions inattendues des tâches si l'utilisateur ne s'attend pas à ce que le programme démarre immédiatement. Un message d'avertissement est affiché pour informer l'utilisateur de cette action.

Vie du projet :

Notre projet s'est déroulé en trois versions notables :

Dans la première version, nous avons réussi à mettre en place la version minimale assez facilement. Cependant, nous avons utilisé des fonctions non recommandées ou interdites pour le projet, notamment celles présentes dans unistd.h. Ces fonctions permettent de remplacer un programme en cours d'exécution par un autre programme. Les différentes fonctions sont les suivantes :

- execl
- execle
- execlp
- execv
- execvp
- execve
- system

Les différences entre ces fonctions résident dans le prototype servant d'interface. Les variantes "l" prennent en paramètre une liste d'arguments, le dernier argument devant être NULL :

- execl : `void execl(const char* path, const char* arg0, const char* arg1, ..., NULL);`

Les variantes "v" prennent en paramètre un tableau d'arguments. Le dernier pointeur du tableau argv[] doit être NULL :

- execv : `void execv(const char* path, char* const argv[]);`

Il est recommandé d'éviter les variantes "p" car elles sont moins sécurisées. Elles permettent la création de portes dérobées sur le système. Pour la même raison, la commande system doit également être évitée. Les variantes "e" fournissent de nouvelles variables d'environnement.

Nous avons donc repris le projet depuis le début pour cette seconde version.
Initialement, nous avions seulement deux fonctions : « planificateur » et « main ».

Rapidement, nous avons rencontré un problème : les exécutions ne se chevauchaient pas si le délai était plus court que le temps d'exécution de la commande. Notre programme attendait la fin de l'exécution de la première itération avant de déclencher le délai, puis la deuxième itération, et ainsi de suite.

C'est lors d'une discussion avec M. Weinberg que nous avons trouvé une métaphore utile pour concevoir notre solution : visualiser le planificateur de tâches comme un canon lançant des boulets. Grâce à cette analogie, nous avons pu aboutir à notre version finale, la version 3.

Nous devions créer notre "canon" (le planificateur) qui lance ses "boulets" (les exécutions). Cela a accéléré notre progression, et grâce à la FAQ, nous avons su quelles fonctions et bibliothèques utiliser. Ainsi, nous avons finalisé notre version minimale avec le « main » récupérant les données de l'utilisateur, le « planificateur » programmant les exécutions, et l' « exécuteur » lançant simplement les commandes.

Ensuite, nous sommes passés aux options supplémentaires après une pause. Implémenter l'option "infini" n'a pas été difficile. Nous avons ajouté une option où l'utilisateur peut entrer "i" pour infini au lieu d'un entier positif pour le nombre d'itérations. Ensuite, nous avons créé une boucle infinie lorsque "i" était spécifié. Cependant, nous avons choisi de représenter l'infini par "-1", ce qui a posé un léger problème : l'utilisateur ne peut pas entrer un nombre négatif d'itérations sauf "-1" pour une exécution infinie. Nous avons jugé que ce n'était pas un problème majeur car cela n'empêchait pas le bon déroulement du programme, et nous l'avons laissé ainsi.

En revanche, l'implémentation de l'option de date a été plus longue car nous avons utilisé la bibliothèque « time », qui a certaines conventions auxquelles nous devions nous adapter. Après beaucoup de débogage pour comprendre pourquoi le calcul du temps d'attente avant le lancement était si long, nous avons remarqué que le code ajoutait une heure à l'heure spécifiée par l'utilisateur. Nous n'avons jamais identifié la raison de ce comportement, mais nous avons contourné le problème en soustrayant une heure à l'heure spécifiée par l'utilisateur, ce qui a permis au code de fonctionner correctement et même de fournir le temps restant en secondes avant le démarrage du planificateur de commandes.

La veille de la remise, certains étudiants ont mentionné qu'il était nécessaire de laisser la main à l'utilisateur s'il y avait des délais d'attente, remettant ainsi en question notre code et nos choix. Après avoir vérifié l'énoncé et constaté qu'il n'y avait aucune spécification en ce sens, nous avons décidé de le faire quand même en rajoutant une fonction de dernière minute.
Il y a seulement un aspect non fonctionnel : l'utilisateur ne reprend pas le contrôle lorsque nous attendons la date programmée. Cependant, dans l'ensemble, nous sommes plutôt satisfaits de notre travail.

Conclusion :

a. Dans le Cadre du Projet

Au début du projet, notre objectif était de concevoir un planificateur de tâches simple mais efficace, offrant à l'utilisateur la possibilité de définir des commandes à exécuter à intervalles réguliers ou à une date précise. Nous sommes parvenus à réaliser cet objectif en développant un programme fonctionnel qui répond aux spécifications énoncées. Notre version finale du planificateur propose des fonctionnalités telles que la programmation d'itérations infinies, la planification à une date spécifique et la gestion des erreurs. Nous sommes plutôt satisfaits de notre réussite, ayant accompli tout ce que nous avions initialement envisagé.

b. Au-delà du Projet

i. Leçon d'Équipe

Ce projet nous a enseigné l'importance de la collaboration et de la communication au sein d'une équipe. En travaillant ensemble sur ce projet de programmation, nous avons pu développer nos compétences en résolution de problèmes, en gestion de projet et en travail d'équipe.

ii. Potentiel d'Application Étendu

Bien que notre planificateur de tâches ait été initialement développé pour répondre à des besoins spécifiques dans le cadre de ce projet, il offre un potentiel d'application étendu dans divers domaines tels que l'automatisation des tâches, la gestion de projet et la surveillance système. En adaptant et en étendant ses fonctionnalités, notre planificateur pourrait être utilisé dans une variété de scénarios et d'environnements informatiques, offrant ainsi une solution polyvalente pour la gestion efficace des tâches planifiées.

iii. L'Aventure Continue

Ce projet a été une expérience enrichissante qui nous a permis de mettre en pratique nos connaissances en programmation et de relever des défis techniques. Nous sommes fiers du résultat obtenu et nous sommes impatients de poursuivre notre aventure dans le domaine de l'informatique en mettant en pratique les compétences et les leçons apprises lors de ce projet. Cette expérience nous a non seulement permis d'approfondir nos connaissances techniques, mais aussi de développer notre capacité à travailler en équipe et à résoudre des problèmes de manière efficace.