UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et génie informatique

RAPPORT DE PROBLÉMATIQUE

Systèmes d'exploitation et architecture des ordinateurs
GRO300

Présenté à
Jean-Philippe Gouin

Présenté par
Équipe 1
Gabriel Aubut – aubg3402
Marc-Olivier Fecteau – fecm0701

Sherbrooke – 21 juillet 2023

TABLE DES MATIERES

LISTE DES FIGURES

LISTE DES TABLEAUX

1. MÉCANISMES DE SYNCHRONISATION

1.1 RÉCEPTION DES NOUVEAUX ÉVÉNEMENTS MOTEURS

Les objets '*data_*' et '*queue_*' sont modifiés par chacun des threads de moteurs. La synchronisation des événements moteurs est gérée par un '*lock_guard*' qui empêche l'accès à ces objets par plus d'un thread à la fois. De sorte, on évite les problèmes de concurrence comme un écrasement des données ou un accès à un objet inexistant. Le '*lock_guard*' permet de s'assurer que le mutex sera déverrouillé après l'exécution de la fonction, ce qui aide à prévenir les problèmes de '*deadlock'*. La figure ci-dessous montre l'implémentation du mécanisme :

```
void RobotDiag::push_event(RobotState new_robot_state) {
    std::lock_guard<std::mutex> lock(mutex_);

    // Conserve toutes les données
    data_.push_back(new_robot_state);

    // Ajoute le dernier événement à la file d'exportation
    queue_.push(new_robot_state);
    cv_.notify_one();   // Signale qu'une nouvelle donnée est disponible
}
```

Figure 1: mécanisme de synchronisation des événements moteurs

1.2 ACCÈS AU FICHIER CSV

L'accès au fichier CSV est géré par un '*unique_lock*' et un *'wait'*. Le *'wait'* bloque le fil d'exécution jusqu'à ce que la variable conditionnelle *'CV_'* (activée par le '*notify_one'* de la figure précédente) soit activée, donc jusqu'à ce qu'une nouvelle donnée soit disponible. Cela réduit les demandes au processeur, car le fil d'exécution vérifie les données lorsqu'elles sont disponibles plutôt qu'en continu. À ce moment, le *'wait'* verrouille le mutex grâce au '*unique-lock*', ce qui empêche les autres fils d'exécution d'accéder à l'objet *'queue_'* pendant que le fil du CSV y fait des modifications. La figure ci-dessous montre l'implémentation du mécanisme :

```
    // Synchronisation et écriture.
    while(run_) // l'utilisation d'un booléen permet d'arrêter le thread
    {
        std::unique_lock<std::mutex> lock(mutex_);
        cv_.wait(lock, [] { return !queue_.empty(); });
        if (!queue_.empty())
        {
            if (queue_.front().id == id_moteur_filtre) // On peut modifier le numéro du moteur désiré dans Qt
            {
                fprintf(out, "%d;%f;%f;%f;%f;\n",queue_.front().id,queue_.front().t,queue_.front().cur_cmd,queue_.front().cur_pos,queue_.front().cur_vel);

            }
            queue_.pop();
        }
    }
    fclose(out);
}
```

Figure 2: mécanisme de synchronisation de l'accès au fichier CSV

2. FILTRAGE DES DONNÉES RÉELLES

La Figure 2 montre la façon dont le filtrage des données est effectué : lorsque le fil d'écriture reçoit le signal qu'une donnée est disponible, il vérifie l'*ID* du moteur (0 : moteur physique, 1 à n : moteurs simulés), qui est spécifié par l'utilisateur dans *main.cpp* (par défaut, l'*ID* est '0'). Cette condition permet de filtrer les données de façon réactive : si les données proviennent du moteur désiré, elles sont écrites dans le fichier CSV. Les données sont ensuite retirées de la file peu importe le moteur de provenance.

3. PERFORMANCE DU PID

Tableau 1: Pseudocode assembleur du PID

Instruction	Opérande 1	Opérande 2	Description	Cycles
LDA	R1	$V_MOTEUR	R1 := mem($V_MOTEUR)	3
LDA	R2	$V_CIBLE	R2 := mem($V_CIBLE)	3
LDA	R3	$INTEGRAL	R3 := mem($INTEGRAL)	3
LDC	R4	0	R4 := 0	2
SUB	R2	R1	R2 := R2 – R1 (e(t))	3
LDA	R1	$E_PREV	R1 := mem($E_PREV)	3
ADD	R4	R2	R4 := R4 + R2	3
ADD	R3	R2	R3 := R3 + R2 (I(t))	3
SUB	R4	R1	R4 := R4 – R1 ($e_d(t)$)	3
STO	$E_PREV	R2	mem($E_PREV) := R2	5
LDA	R1	$KP	R1 := mem($KP)	3
STO	$INTEGRAL	R3	mem($INTEGRAL) := R3	5
MUL	R2	R1	R2 := R2 * R1	3
LDA	R1	$KI	R1 := mem($KI)	3
MUL	R3	R1	R3 := R3 * R1	3
LDA	R1	$KD	R1 := mem($KD)	3
ADD	R2	R3	R2 := R2 + R3	3
MUL	R4	R1	R4 := R4 * R1	3
ADD	R2	R4	R2 := R2 + R4 (u(t))	3
STO	$CMD	R2	mem($CMD) := R2	5

Le nombre total de cycles serait de 62. Toutefois, ce nombre n'inclut pas les commandes pour éviter les aléas de contrôle, et n'inclut pas l'initialisation de k_P, k_I, et k_D, car il est supposé que ces valeurs sont initialisées ailleurs dans le code, et sont déjà en mémoire lorsque la fonction est appelée (autrement, il y aurait 21 cycles supplémentaires, pour un total de 83 cycles). Le temps de processeur utilisateur (*user CPU time*) est obtenu avec l'équation suivante :

$$uCPU\ time = \#cycles * t_{cycle} = \#cycles * \frac{f_{CPU}}{CPI_{CPU}} \quad (1)$$

Pour calculer le $uCPU\ time$, on devrait connaître la fréquence d'opération du *CPU*, ainsi que son *CPI*.

4. Enregistrement de l'état du moteur

Le moteur sélectionné pour la démonstration était le moteur physique. La vitesse désirée était de 0,7 m/s (le seuil était de 0,01). La Figure 3 montre que le moteur atteint éventuellement la vitesse désirée, mais que le temps requis pour atteindre cette vitesse est irraisonnable. Aussi, bien que le laps de temps soit assez court, on remarque sur cette même figure que lorsque la vitesse désirée est atteinte, le moteur reste à cette vitesse. La série 'cmd_robot' permet de visualiser la commande (en valeur absolue) envoyée au moteur, et permet de vérifier que le moteur ne reçoit pas de commande en dehors de l'intervalle $[-1, 1]$.

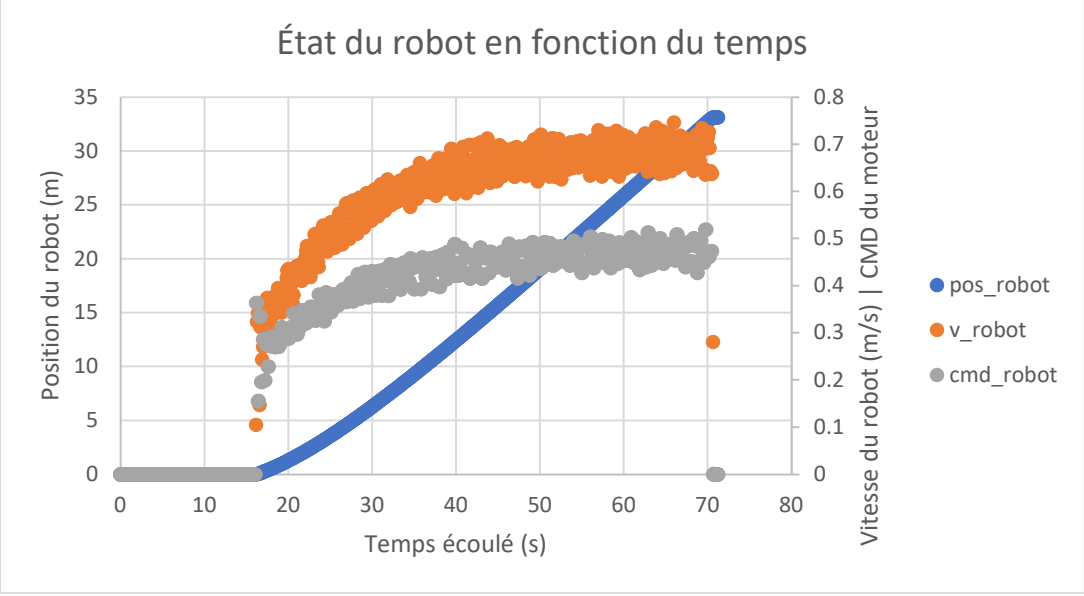

Figure 3: Graphique de l'état du moteur

5. CONCLUSION

Considérant que :

- Tous les moteurs, réels et simulés, opèrent sur un fil différent ;
- L'écriture dans le fichier CSV se fait dans un fil séparé ;
- Le programme filtre les données de sorte que seulement les données du moteur sélectionné ne soient dans le fichier CSV ;
- Des mécanismes de synchronisation adéquats sont implémentés à tous les endroits où un aléa (*e.g.* écrasement de données, accès concurrentiel à un espace mémoire) a été identifié ;
- Le pseudocode assembleur PID fonctionne théoriquement ;
- L'état du moteur est enregistré de façon adéquate ;
- La loi de contrôle est vérifiée, bien que ses réglages ne soient pas optimaux.

La solution proposée est jugée adéquate.