INF3610 – Systèmes embarqués

Automne 2024

TP No. 1 (Partie 3)

Groupe 02

[redacted]

Soumis à : [redacted]

Vendredi 15 Novembre 2024

Questions pour le rapport

Question 1 Soit les 2 cas suivants qui abstrait le fonctionnement de TaskGenerate :

Cas 1)

- *Generation d'une rafale (burst)*
- *while(!*ack);*
- **req = 1; xil_printf("Task_Generate Fin rafale no: %d completee avec %d paquets\n", *burst_no, *number_of_packets);*
- *while(*ack);*
- **req = 0;*
- *burst_number++;*

Cas 2)

- *while(!*ack);*
- **req = 1;*
- *Génération d'une rafale*
- *while(*ack);*
- **req = 0;*
- *burst_number++;*

- *xil_printf("Task_Generate Fin rafale no: %d completee avec %d paquets\n", *burst_no, *number_of_packets);*

Lequel des cas s'applique à votre implémentation et quel est son avantage par rapport à l'autre cas.

Dans le cadre de notre implémentation, **le Cas 1** s'applique pleinement au fonctionnement de la tâche **`TaskGenerate`**. Cette décision repose sur l'analyse détaillée du flux d'exécution de la tâche telle que implémentée dans le fichier **`generator.c`**.

1. Correspondance avec l'Implémentation

Description du Cas 1 :

1. **Génération d'une rafale (burst)**
2. **Attente de l'acknowledgement (`while(!*ack);`)**
3. **Envoi du signal de requête (`*req = 1;`)**
4. **Affichage des informations sur la rafale générée**
5. **Attente de la réinitialisation de l'acknowledgement (`while(*ack);`)**
6. **Réinitialisation du signal de requête (`*req = 0;`)**
7. **Incrémentation du numéro de rafale (`burst_number++;`)**

La séquence des opérations dans `TaskGenerate` suit rigoureusement le **Cas 1**. La tâche commence par générer une rafale complète de paquets avant d'attendre l'acknowledgement (`*ack`). Une fois la génération terminée, elle envoie le signal de requête (`*req = 1;`), affiche

les informations relatives à la rafale générée, puis attend la réinitialisation de l'acknowledgement avant de réinitialiser le signal de requête et d'incrémenter le numéro de rafale.

2. Avantages du Cas 1 par Rapport au Cas 2

a. Séparation Claire des Phases de Génération et de Communication

Le **Cas 1** assure une séparation nette entre la phase de génération des paquets et la phase de communication avec `TaskComputing`. En générant d'abord tous les paquets nécessaires avant d'envoyer le signal de requête, la tâche garantit que toutes les données sont prêtes et cohérentes avant qu'elles ne soient traitées par une autre tâche ou un autre cœur. Cela réduit le risque d'incohérences ou d'erreurs dues à des paquets partiellement générés.

b. Prévention des Conditions de Course

En envoyant le signal de requête (`*req = 1;`) uniquement après avoir complètement généré la rafale, le **Cas 1** évite les conditions de course où `TaskComputing` pourrait commencer à traiter des paquets avant que leur génération ne soit terminée. Dans le **Cas 2**, où le signal de requête est envoyé avant la génération complète des paquets, il existe un risque que `TaskComputing` interagit avec des données incomplètes ou corrompues, entraînant des comportements indéfinis ou des erreurs.

c. Amélioration de la Fiabilité et de la Cohérence des Données

Le **Cas 1** renforce la fiabilité du système en garantissant que les paquets sont entièrement générés et valides avant qu'ils ne soient signalés pour traitement. Cela assure une cohérence des données entre les tâches ou les cœurs, facilitant ainsi la maintenance et le débogage du système.

d. Gestion Optimisée du Watchdog Timer

Dans notre implémentation, le watchdog timer est réinitialisé après avoir reçu l'acknowledgement de `TaskComputing`, ce qui correspond au **Cas 1**. Cette approche garantit que le watchdog timer ne se déclenche que lorsqu'une rafale a été correctement traitée, offrant une surveillance plus précise de la santé de `TaskComputing`. Dans le **Cas 2**, où le signal de requête est envoyé avant la génération complète, le watchdog timer pourrait être réinitialisé prématurément, réduisant ainsi son efficacité à détecter les pannes.

<u>Question 2</u> En principe le fait d'avoir déplacé TaskGenerate sur le core1 devrait permettre aux tâches restantes sur le core 0 (dont TaskQueuing) de s'exécuter plus rapidement et donc d'améliorer les pires temps vidéo, audio et autres. Pourtant, l'augmentation de la variable delai_pour_vider_les_fifos_msec pour respecter le 50% de remplissage des fifos lors de la manipulation 3.1 semble indiquer le contraire. Pourquoi selon vous? Expliquez en détails.
N.B. Si parfois vous n'avez pas eu besoin d'augmenter la variable delai_pour_vider_les_fifos_msec et que vous êtes convaincu de son minimum, expliquez en ce sens.

Dans notre implémentation, bien que déplacer `TaskGenerate` sur le cœur 1 vise à alléger la charge du cœur 0 et à améliorer les performances globales, plusieurs facteurs ont conduit à l'augmentation de `delai_pour_vider_les_fifos_msec` pour maintenir un remplissage optimal des FIFO. Voici les raisons détaillées :

1. **Surcharge de Synchronisation Inter-Cœurs :**
    - **Communication Latente :** La synchronisation entre les cœurs via des variables partagées (`req`, `ack`, `shutdown_flag`) introduit une latence supplémentaire. Chaque accès nécessite des mécanismes de synchronisation, augmentant le temps de traitement global.
    - **Attente Active :** Les boucles d'attente (`while(!*ack);` et `while(*ack);`) peuvent provoquer une utilisation intensive du CPU pour vérifier constamment l'état des signaux, ralentissant ainsi l'exécution des tâches.
2. **Contention sur les Ressources Mémoire Partagées :**
    - **Accès Concurrent :** Les deux cœurs accèdent simultanément à la mémoire partagée pour lire et écrire les signaux. Même avec le cache désactivé, cela peut entraîner des conflits d'accès, réduisant l'efficacité globale.
    - **Bande Passante Limité :** La bande passante mémoire partagée peut devenir un goulot d'étranglement lorsque plusieurs cœurs tentent d'accéder fréquemment aux mêmes adresses mémoire.
3. **Impact de la Désactivation des Caches :**
    - **Performance Réduite :** Désactiver les caches (`Xil_DCacheDisable()`) augmente la latence des accès mémoire, car chaque lecture ou écriture doit passer par la mémoire principale. Cela ralentit les opérations de synchronisation et de communication entre les cœurs.
    - **Absence d'Optimisation :** Sans caches, les cœurs ne bénéficient pas des gains de performance liés à la réduction des accès mémoire fréquents.
4. **Augmentation de la Complexité des Tâches :**
    - **Gestion des Signaux :** La gestion des signaux `req` et `ack` complique le flux des tâches, introduisant des points de synchronisation supplémentaires qui peuvent ralentir l'exécution.
    - **Interdépendance des Tâches :** Les tâches sur différents cœurs deviennent plus interdépendantes, ce qui peut limiter les optimisations possibles et introduire des délais imprévus.
5. **Configuration des Delais pour Vider les FIFO :**

- **Réajustement Nécessaire :** Pour compenser les latences accrues dues à la synchronisation et à la contention, il a été nécessaire d'augmenter `delai_pour_vider_les_fifos_msec`. Cela assure que les FIFO ne se remplissent pas trop rapidement malgré les ralentissements.
- **Maintien du Taux de Remplissage :** L'augmentation du délai permet de réguler le flux des paquets, évitant une surcharge des FIFO qui pourrait entraîner des rejets de paquets ou une augmentation des temps de traitement.

Conclusion

Bien que déplacer `TaskGenerate` sur le cœur 1 ait théoriquement dû libérer des ressources sur le cœur 0 et améliorer les performances, les défis liés à la synchronisation inter-cœurs, la contention des ressources mémoire partagées, et la désactivation des caches ont contrebalancé ces avantages. L'augmentation de `delai_pour_vider_les_fifos_msec` était nécessaire pour maintenir un taux de remplissage optimal des FIFO face à ces nouvelles latences introduites.

Question 3 À propos de la minuterie watchdog

a) Expliquez comment vous avez implémenté votre watchdog. Plus précisément, expliquez les paramètres de OSTmrCreate, justifiez le mode de fonctionnement du watchdog et décrire comment la fonction de callback réalise le travail de stopper TaskGenerate.

À propos de la minuterie watchdog

a) Implémentation du Watchdog Timer

Dans notre implémentation, le watchdog timer a été intégré pour surveiller le bon fonctionnement de la tâche `TaskGenerate` sur le cœur 1. Voici les détails de son implémentation :

Création du Watchdog Timer avec `OSTmrCreate`
La fonction `OSTmrCreate` est utilisée pour créer et configurer le watchdog timer. Les paramètres passés à cette fonction sont les suivants :

```
OSTmrCreate(
     &WatchdogTmr,                          // Pointeur vers la
structure du timer
    "Watchdog Timer",                 // Nom du timer
    WATCHDOG_TIMEOUT_TICKS,           // Délai initial en ticks
     0,                                     // Périodicité (0 pour
one-shot)
     OS_OPT_TMR_ONE_SHOT,                // Options du timer (mode
one-shot)
    ShutdownTaskGenerateFct,          // Fonction de callback
    NULL,                             // Argument pour la callback
    &err                              // Variable pour les erreurs
);
```

1.
    - **`&WatchdogTmr`** : Pointeur vers la structure du timer `WatchdogTmr`.
    - **`"Watchdog Timer"`** : Nom descriptif du timer pour faciliter le débogage.
    - **`WATCHDOG_TIMEOUT_TICKS`** : Définition du délai d'expiration du watchdog en ticks, calculé comme `WATCHDOG_TIMEOUT_SEC * OS_CFG_TICK_RATE_HZ` (par exemple, 30 secondes).
    - **`0`** : Périodicité mise à zéro pour indiquer un mode one-shot, c'est-à-dire que le timer ne se répète pas automatiquement après expiration.
    - **`OS_OPT_TMR_ONE_SHOT`** : Option spécifiant que le timer fonctionne en mode one-shot.
    - **`ShutdownTaskGenerateFct`** : Fonction de callback appelée lorsque le timer expire.
    - **`NULL`** : Aucun argument supplémentaire n'est passé à la callback.
    - **`&err`** : Variable pour capturer d'éventuelles erreurs lors de la création du timer.
2. **Justification du Mode de Fonctionnement (One-Shot)**
   Le mode **one-shot** a été choisi car le watchdog timer doit surveiller des événements spécifiques sans se réinitialiser automatiquement. Chaque fois que la condition surveillée est respectée (par exemple, réception d'un signal `ack`), le timer est réinitialisé manuellement. Cela permet de détecter précisément les périodes d'inactivité ou de dysfonctionnement sans interférence de réinitialisations automatiques.

Fonction de Callback `ShutdownTaskGenerateFct`
La fonction de callback `ShutdownTaskGenerateFct` est déclenchée lorsque le watchdog timer expire, indiquant que la tâche `TaskGenerate` n'a pas répondu dans le délai imparti. Voici son implémentation simplifiée :

```
void ShutdownTaskGenerateFct(OS_TMR *p_tmr, void *p_arg)

{

       xil_printf("Watchdog timer expired. Shutting down core 1.\n");

    // Arrêter les tâches et libérer les ressources

    delete_tasks();

}
```

3. **Description du Fonctionnement :**
    - **Notification de l'Expiration :** La fonction commence par afficher un message indiquant que le watchdog timer a expiré.
    - **Arrêt des Tâches :** Elle appelle la fonction `delete_tasks()` qui supprime les tâches critiques, notamment `TaskGenerate`, assurant ainsi que le cœur 1 s'arrête proprement en cas de dysfonctionnement.
    - **Libération des Ressources :** En plus de stopper les tâches, cette fonction peut être étendue pour libérer d'autres ressources ou effectuer des opérations de nettoyage si nécessaire.
4. **Flux d'Exécution :**
    - **Création et Démarrage du Timer :** Dans la tâche `TaskGenerate`, le watchdog timer est créé et démarré au début de chaque itération de génération de rafale.
    - **Réinitialisation du Timer :** Une fois que `TaskGenerate` reçoit l'acknowledgement (`ack`) de `TaskComputing`, il réinitialise le watchdog timer pour commencer un nouveau délai.
    - **Expiration du Timer :** Si `TaskGenerate` ne parvient pas à réinitialiser le watchdog timer dans le délai imparti (30 secondes), la fonction de callback est appelée, déclenchant ainsi l'arrêt du cœur 1.

b) Précision du Watchdog Timer et Solution AXI Timer-based Watchdog

Limitation du Watchdog Actuel :

Comme mentionné en classe, une des limitations du watchdog timer basé sur `OSTmr` réside dans sa **précision limitée par le tick rate** de l'horloge système. Par exemple, si le tick rate est de 100

Hz (10 ms par tick), le délai d'expiration ne peut être réglé qu'en multiples de 10 ms. Cela peut introduire une imprécision, surtout pour des délais courts ou critiques.

Solution Offerte par le Composant AXI Timer-based Watchdog de Xilinx :

Le **AXI Timer-based Watchdog Timer** fourni par la librairie Xilinx présente des avantages significatifs en termes de précision par rapport au watchdog basé sur les ticks d'`OSTmr`. Voici pourquoi :

1. **Précision Haute :**
    - **Fréquence de l'Horloge AXI Timer :** L'AXI Timer peut fonctionner à une fréquence beaucoup plus élevée que le tick rate du système d'exploitation, permettant des délais d'expiration plus précis et plus courts.
    - **Granularité Fine :** Grâce à une granularité fine des compteurs matériels, l'AXI Timer permet de configurer des délais d'expiration avec une précision bien supérieure, souvent dans la gamme des microsecondes ou nanosecondes selon la configuration matérielle.
2. **Indépendance du Tick Rate de l'OS :**
    - **Fonctionnement Asynchrone :** L'AXI Timer fonctionne indépendamment du tick rate de l'OS, ce qui signifie que sa précision n'est pas limitée par la fréquence des ticks logiciels.
    - **Minimisation des Délais Logiciels :** Les délais d'expiration sont gérés directement par le matériel, réduisant la latence introduite par les interruptions et la gestion logicielle des ticks.
3. **Réduction de la Charge Logicielle :**
    - **Moins de Surcharge CPU :** Étant géré par le matériel, l'AXI Timer-based Watchdog réduit la charge sur le CPU, améliorant ainsi les performances globales du système.
    - **Gestion Automatique des Interruptions :** Le matériel peut automatiquement déclencher des interruptions ou des callbacks sans nécessiter une gestion intensive par le logiciel.
4. **Fiabilité Accrue :**
    - **Moins Sujet aux Variations Logicielles :** Contrairement aux timers logiciels, le watchdog matériel est moins susceptible d'être affecté par les variations de charge logicielle ou les retards d'exécution.
    - **Surveillance Continue :** L'AXI Timer peut surveiller de manière continue et fiable les conditions de fonctionnement sans dépendre de l'état des tâches logicielles.

Justification :

En utilisant le **AXI Timer-based Watchdog Timer**, nous bénéficions d'une **précision améliorée** et d'une **fiabilité accrue**, essentielles pour des applications critiques où les délais de réponse doivent être strictement respectés. Cela permet de détecter plus rapidement et précisément les dysfonctionnements, assurant ainsi une meilleure gestion des erreurs et une sécurité renforcée du système.

Conclusion :

Le watchdog timer actuel basé sur `OSTmr` est fonctionnel mais limité par la précision du tick rate de l'OS. L'adoption d'un **AXI Timer-based Watchdog Timer** offrirait une solution plus précise et fiable, répondant mieux aux exigences des applications temps réel et réduisant les risques liés aux délais d'expiration imprécis.