Travaux Pratiques - Travaux Dirigés - Systèmes multitâches

Nous ne cherchons pas à faire dans ce TP une application temps réel. Les APIs que vous allez manipuler, les phénomènes que vous allez observer et la conception que vous allez mettre en œuvre dans ce TP seraient reproductibles sur un système d'exploitation dit « Temps Réel ».

Objectifs

Les objectifs de ce TP sont multiples :

- Concevoir des applications multitâche (non temps réel) avec une approche dirigée par les événements
- Manipuler les mécanismes POSIX d'un système d'exploitation Linux et les mécanismes C11
- Concevoir une application multitâche avec une approche dirigée par le temps

Sources

Le code source pour ce tp est présent à l'adresse suivante :

https://github.com/fthomasfr/multitasking_training_practical_work

Pour le récupérer vous pouvez le télécharger directement ou utiliser git :

```
git clone https://github.com/fthomasfr/multitasking_training_practical_work.git
```

L'ensemble des informations et codes d'exemples concernant la programmation multitâche sont disponibles dans votre cours.

Pour compiler votre programme, un makefile est proposé. Il possède les règles suivantes:

- `make posix` : compile le programme incluant le fichier `acquisitionManagerPosix.c` utilisé en question 7
- `make runposix` : compile et execute le programme incluant le fichier `acquisitionManagerPosix.c` utilisé en question 7
- `make atomic` : compile le programme incluant le fichier `acquisitionAtomic.c` utilisé en question 10
- `make runatomic` : compile le programme incluant le fichier `acquisitionAtomic.c` utilisé en question 10
- `make testandset` : compile le programme incluant le fichier `acquisitionTestAndSet.c` utilisé en question 13
- `make runtestandset` : compile le programme incluant le fichier `acquisitionTestAndSet.c` utilisé en question 13
- `make clean` : supprime les executables et l'ensemble des fichiers et artefacts de compilation

Exercice Principal

Comme le montre le schéma d'architecture système de la figure 1, notre logiciel réalise l'acquisition de quatre entrées numériques asynchrones auprès d'un composant logiciel externe nommé `SensorManager`. Notre logiciel possède une sortie numérique et une sortie de diagnostic connectées à un `Display`. Le `SensorManager` met à disposition pour chaque entrée un tableau de 256 entiers et un checksum modélisés par le type de donnée `MSG_BLOCK`. La sortie du numérique est caractérisée par un tableau de 256 entiers et un checksum modélisés par le même type de donnée `MSG_BLOCK`. La sortie diagnostic est une chaine de caractères à destination du terminal.

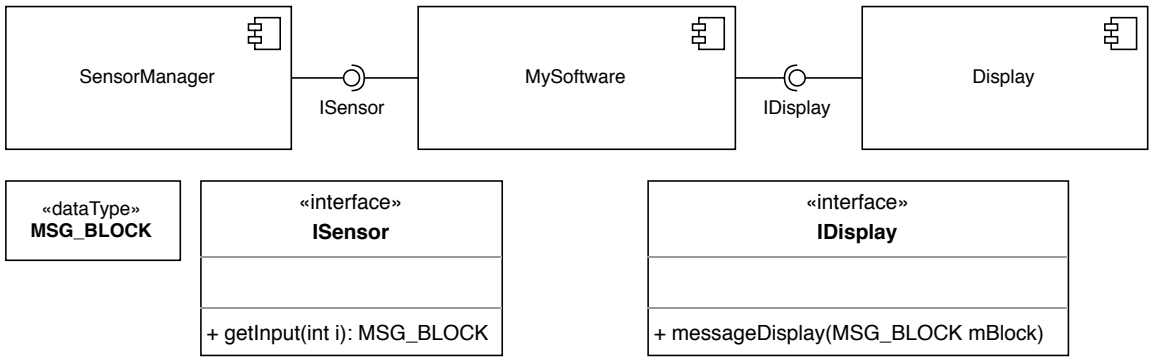

Figure 1: Architecture Système La description de l'architecture système dans laquelle notre logiciel est intégré.

Depuis cette architecture système les exigences suivantes ont été assignées à `MySoftware` :

- **Exigence 1** : Le logiciel doit acquérir quatre entrées et produire en sortie le cumul des entrées dès qu'une entrée est acquise. Le cumul correspond à la production d'un tableau de 256 entiers dont le ieme élément de ce tableau est la somme des iemes éléments des tableaux d'entrée.
- **Exigence 2** : Le logiciel doit acquérir toutes les données d'entrées.
- **Exigence 3** : Le logiciel doit garantir que les données d'entrée sont correctement formées avant de les sommer.
- **Exigence 4** : Le logiciel doit sommer et produire une sortie au plus vite, c'est-à-dire dès qu'une entrée est présente sans attendre une nouvelle valeur sur chacune des entrées.
- **Exigence 5** : Le logiciel doit produire sur la sortie diagnostic pour chaque opération de cumul, combien d'entrées ont été acquises, combien ont été sommées et combien restent à sommer.

Un architecte logiciel a décrit dans un Software Architecture Document (SAD), l'architecture de `MySoftware` à mettre en œuvre :

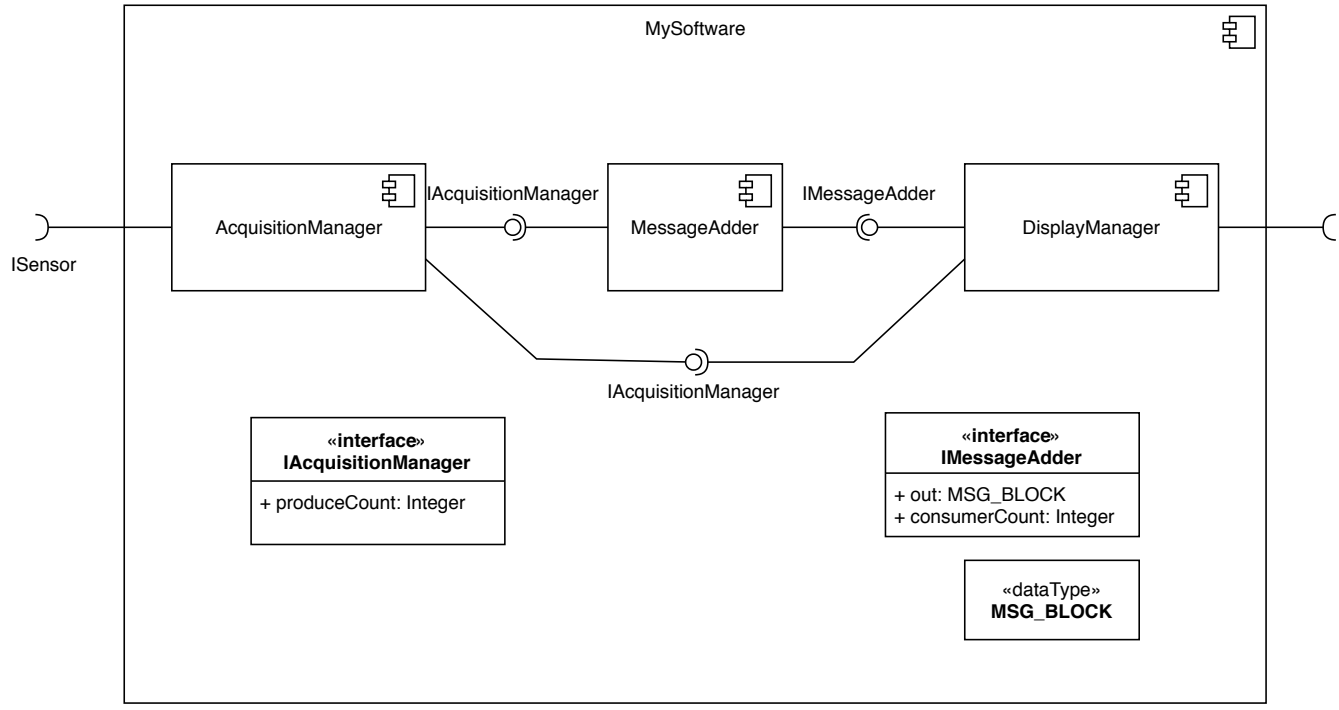

Figure 2: Architecture Logicielle La description de l'architecture logicielle à mettre en oeuvre.

`AcquisitionManager` acquiert les entrées depuis l'interface `ISensor`. `MessageAdder` somme ces entrées. `DisplayManager` pilote la sortie.

1. Complétez cette architecture logicielle en allouant les exigences de la spécification précédente sur les composants de `MySoftware`.

L'objectif de ce TP est de décrire l'architecture détaillée multitâche et de proposer plusieurs implémentations.

Conception Détaillée en utilisant une approche dirigée par les événements

2. Une approche dirigée par les événements est-elle une approche synchrone ou asynchrone ?

Plusieurs conceptions détaillées peuvent exister. Nous proposons de guider votre conception en suivant les choix de conception suivants:

- Utilisation de tâches où la mémoire est partagée entre les tâches;
- Partage d'un tableau de messages. Un message est une entrée acquise;
- Utilisation d'événements sans transmission de données pour synchroniser les tâches;
- Utilisation d'un checksum pour satisfaire l'exigence 3;
- L'utilisation des librairies de messages simulant les entrées (`SensorManager`) et l'affichage (`Display`). La fonction `getInput` permet de simuler la production d'une entrée. La fonction `messageDisplay` permet d'afficher le message.

Nous proposons une architecture détaillée préliminaire en figure 2 et un squelette d'implémentation de cette architecture dans les `.h` et `.c` fournis. Etudiez le squelette d'implémentation `.h` et `.c` fournis et l'architecture logicielle détaillée ci-dessous pour en comprendre les structures de données et les APIs.

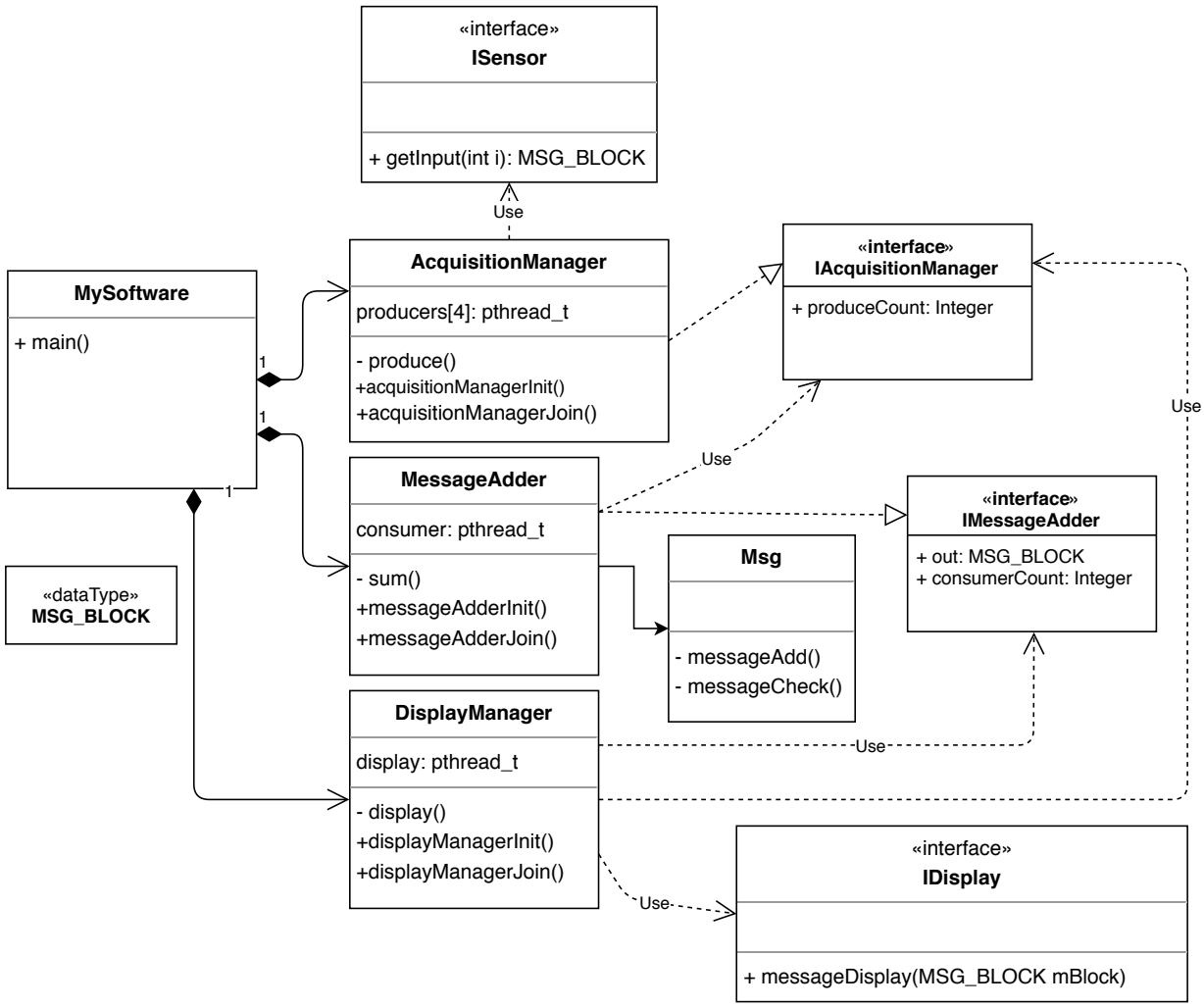

Figure 2: Architecture Logicielle Détaillée La description de l'architecture logicielle détaillée mise en oeuvre.

3. Pourquoi certaines variables sont-elles considérées comme des variables C volatiles dans l'implémentation proposée ?
4. A ce stade de l'implémentation squelette proposée, combien y a-t'il de processus (`process`) et de fil d'exécution (`thread`) POSIX dans ce programme ?
5. Complétez cette architecture logicielle détaillée par l'analyse des tâches à mettre en oeuvre, des échanges de données et des synchronisations entre les tâches en satisfaisant les choix de conception précédents.

Cette conception devra donc détailler l'architecture dynamique et par conséquent utiliser des diagrammes de séquences pour expliquer et démontrer la causalité des traitements. Vous devrez mettre à jour les diagrammes de classes pour y ajouter les APIs dont vous avez besoin.

Pour rappel, nous souhaitons suivre les bonnes pratiques de conception notamment celle "Acquire and release synchronization primitives in the same module, at the same level of abstraction" vue en cours. Si vous avez besoins de sémaphore et mutex par exemple, cela implique de créer des accesseurs pour limiter l'utilisation de ces sémaphores et mutex au seul module C qui les déclare. Il vous faudra à minima les fonctions `incrementProducerCount` et `getProducerCount` dans l'`AcquisitionManager` pour la suite du TP. La fonction `incrementProducerCount` pourra rester locale au module. `getProducerCount` devra être publique.

6. Nous avons orienté l'achitecture détaillée avec l'utilisation des tâches. Nous aurions pu utiliser des processus POSIX. Citez les caractéristiques intéressantes des processus pour une conception orientée « sûreté de fonctionnement » ? Citez également celles qui ne sont pas satisfaites ? Auriez-vous pu utiliser les mêmes choix de conception que ceux-ci-dessus ?

Implémentation dirigée par les événements

7. Implémentez votre conception (Implémentation + Exécution) et montrez un résultat d'exécution.
8. Mesurez à l'aide du script `systemtap` fournit le temps moyen d'exécution de ces fonctions (scripts et readme fournis dans le répertoire `probe`)

Pour aller plus loin en conception et implémentation dirigées par les événements

9. Proposez une solution pour protéger de manière efficace entre les tâches et sans utilisez des apis POSIX, le compteur permettant compter le nombre de messages produits ?
10. Implémentez cette solution dans la méthode `incrementProducerCount` et `getProducerCount` et montrez un résultat d'exécution.
11. Mesurez à l'aide du script systemtap fournit le temps moyen d'exécution de ces fonctions (scripts et readme fournis dans le répertoire `probe`)
12. Proposez une autre solution pour que ces méthodes `incrementProducerCount` et `getProducerCount` en vous basant sur la méthode `atomic_compare_exchange_weak` ?
13. Implémentez cette solution dans la méthode `incrementProducerCount` et `getProducerCount` en définissant deux méthodes `pCountLockTake()` et `pCountLockRelease()`. Montrez un résultat d'exécution.
14. Mesurez à l'aide du script `systemtap` fournit le temps moyen d'exécution de ces fonctions (scripts et readme fournis dans le répertoire `probe`)
15. Concluez sur vos différentes mesures.

Conception détaillée en utilisant une approche dirigée par le temps

16. Une approche dirigée par le temps est-elle une approche synchrone ou asynchrone ?

Nous souhaitons ajouter des exigences liées au temps et définir une conception utilisant le modèle Psy vue en cours (Logical Execution Model). Les exigences supplémentaires sont les suivantes:

- **Exigence 6** : Le logiciel doit acquérir les entrées toutes les 100 ms.
- **Exigence 7** : Le logiciel doit produire en sortie le cumul des entrées toutes les 600 ms.

17. Proposez sur un diagramme de temps, une conception dirigée par le temps en utilisant une ligne horizontale par horloge et une ligne par thread. Les ticks d'horloges seront des points. Les points de synchronisation temporels seront représentés par des lignes verticales. Les messages par des enveloppes à l'instant ou ces messages sont visibles des consommateurs (voir exemple vu en cours).