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

- Manipuler 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.

Préambule

L'objectif de ce préambule est de vous faire découvrir la création d'une tâche et d'un sémaphore en utilisant les APIs POSIX. Ces APIs seront utilisées dans l'exercice principal.

Pour compiler le programme de ce preambule, un makefile est proposé avec les règles suivantes:

- `make preambule` permetttant de compiler le programme de préambile incluant le fichier `preambule.c`
- `make runpreambule` : compile et execute le programme de préambile incluant le fichier `preambule.c`
- `make clean` : supprime les executables et l'ensemble des fichiers et artefacts de compilation

1. Ouvrez le fichier `preambule.c` avec l'éditeur de votre choix.

Pour visualiser la documentation d'une API POSIX man [nom de la méthode] dans une ligne de commande

A l'aide de la partie 3 de votre cours présentant les APIs POSIX:

2. Identifiez dans le programe le nom du thread.
3. Identifiez dans le programe le nom du sémaphore.
4. Identifiez dans le programe le nom du mutex.
5. Identifiez la création du thread.
6. Identifiez dans le programe le point d'entré du thread.
7. Identifiez l'attente de la fin du thread pour terminer le processus courant.
8. Identifiez le thread par défaut du processus courant.

9. Complétez le diagramme ci-dessous expliquant la conception détaillée de ce processus et de ce thread en remplaçant les `[TODO]`. Les sources de ce diagramme à compléter sont rangées dans le premier onglet du fichier diagrams/conception.drawio editable avec le programme en ligne draw.io. Vous devez ensuite exporter votre diagramme pour l'intégrer à votre compte rendu. Seul le diagramme dans le compte rendu compte.
10. En partant de l'hypothèse qu'il n'existe pas d'exigences temporelles sur ce programme, sa conception est-elle complète ?

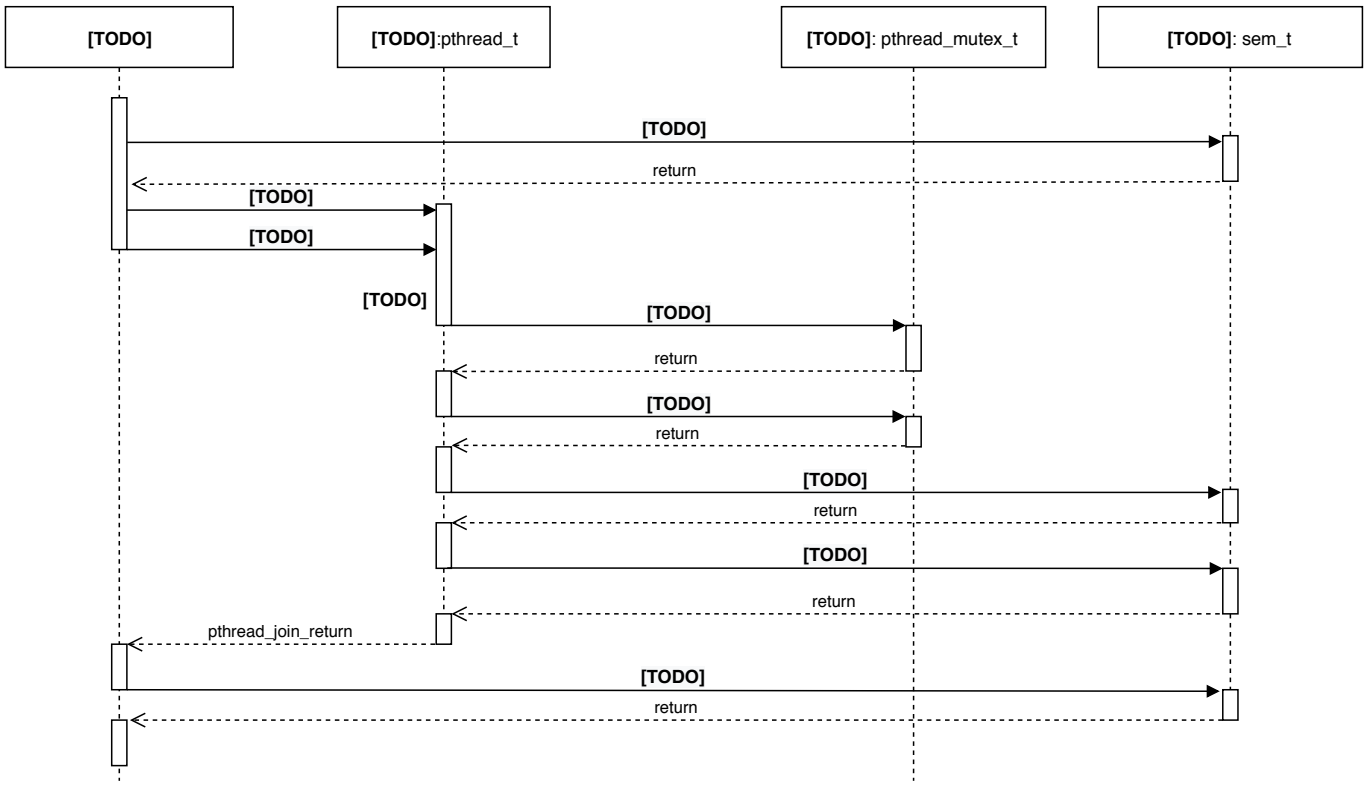

Figure 1: Conception détaillée du processus de découverte des APIs POSIX. La description de l'architecture dynamique du processus de découverte des APIs POSIX.

Premier exercice (Exercice Principal)

Comme le montre le schéma d'architecture système de la figure 1, nous cherchons à développer un accumulateur nommé `MultitaskingAccumulator` qui réalise l'acquisition de quatre entrées numériques asynchrones auprès d'un composant logiciel externe nommé `SensorManager` et qui somme ces entrées. Il produit une sortie numérique correspondante à cette somme et une sortie de diagnostic. Ces sorties sont connectées à un `Display` pour affichage.

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 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.

La fonction `getInput` de `SensorManager` permet de simuler la production d'une entrée. La fonction `messageDisplay` de `Display` permet d'afficher le message. De même, la fonction `print` permet d'afficher le nombre de messages produits, consommés et la différence entre les deux.

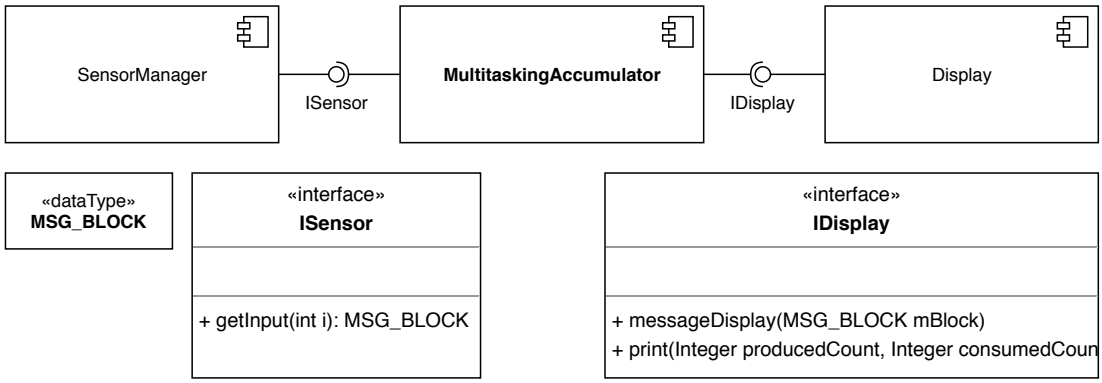

Figure 1: Architecture Système. La description de l'architecture système dans laquelle notre logiciel est intégré.

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

- **Exigence 1** : MultitaskingAccumulator 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** : MultitaskingAccumulator doit acquérir toutes les données d'entrées.
- **Exigence 3** : MultitaskingAccumulator doit garantir que les données d'entrée sont correctement formées avant de les sommer.
- **Exigence 4** : MultitaskingAccumulator 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.

Note: Pour l'exigence 1, nous cherchons à développer un accumulateur. Il est donc inutile d'attendre les quatres entrées pour produire une somme. Au fur et à mesure des valeurs en entrée, le logiciel doit sommer et cela indépendamment de l'entrée. Nous devons sommer deux valeurs succéssives de la première entrée si elles arrivent avant une valeur sur la deuxième entrée par exemple.

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

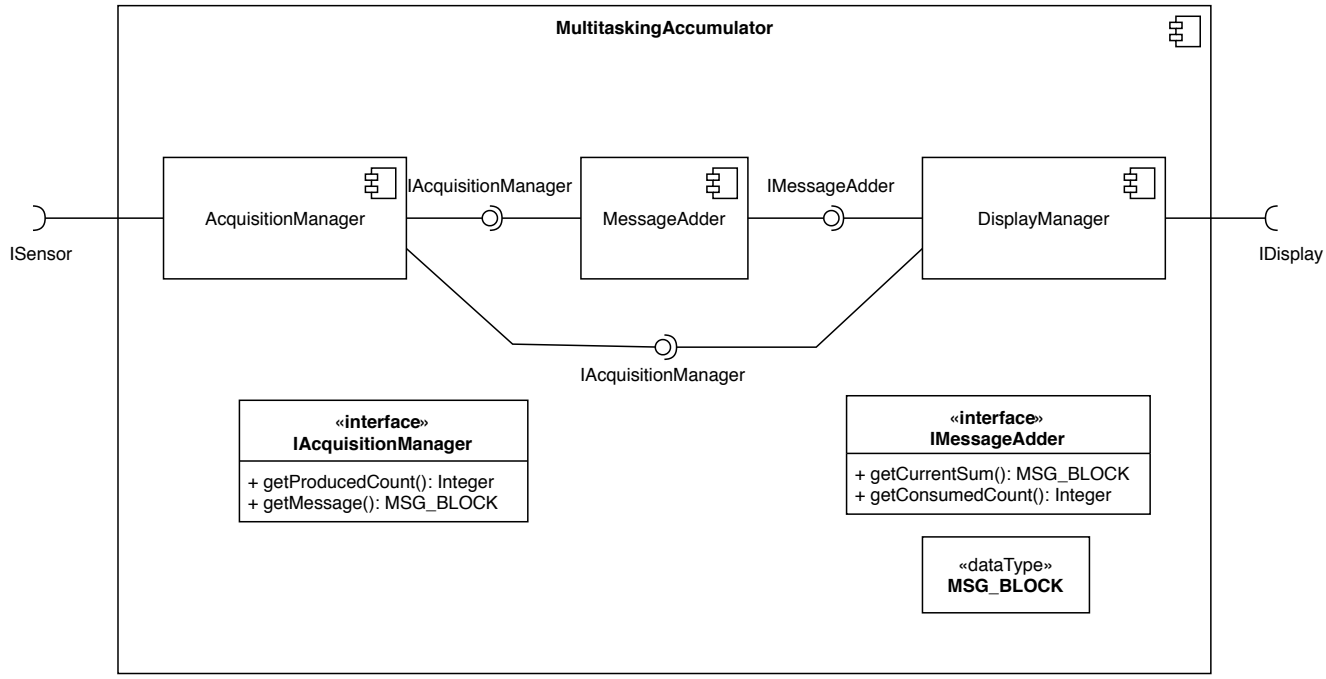

Figure 2: Architecture Logicielle de MultitaskingAccumulator. La description de l'architecture logicielle à mettre en oeuvre.

`AcquisitionManager` acquiert les entrées depuis l'interface `ISensor`. `MessageAdder` somme ces entrées. `DisplayManager` pilote la sortie. `MessageAdder` obtient un nouveau message produit par la méthode `getMessage` d'`AcquisitionManager`.

`MessageAdder` et `DisplayManager` peuvent obtenir le nombre de messages produits par la méthode `getProducedMessage`. De même, `DisplayManager` obtient le nombre de messages consommés et la somme courante par les méthodes `getConsumedCount` et `getCurrentSum` fournies par `MessageAdder`.

1. Complétez cette architecture logicielle en allouant les exigences de la spécification précédente sur les composants de `MultitaskingAccumulator`. Un exemple d'allocation à compléter et à modifier est illustré en figure 3.

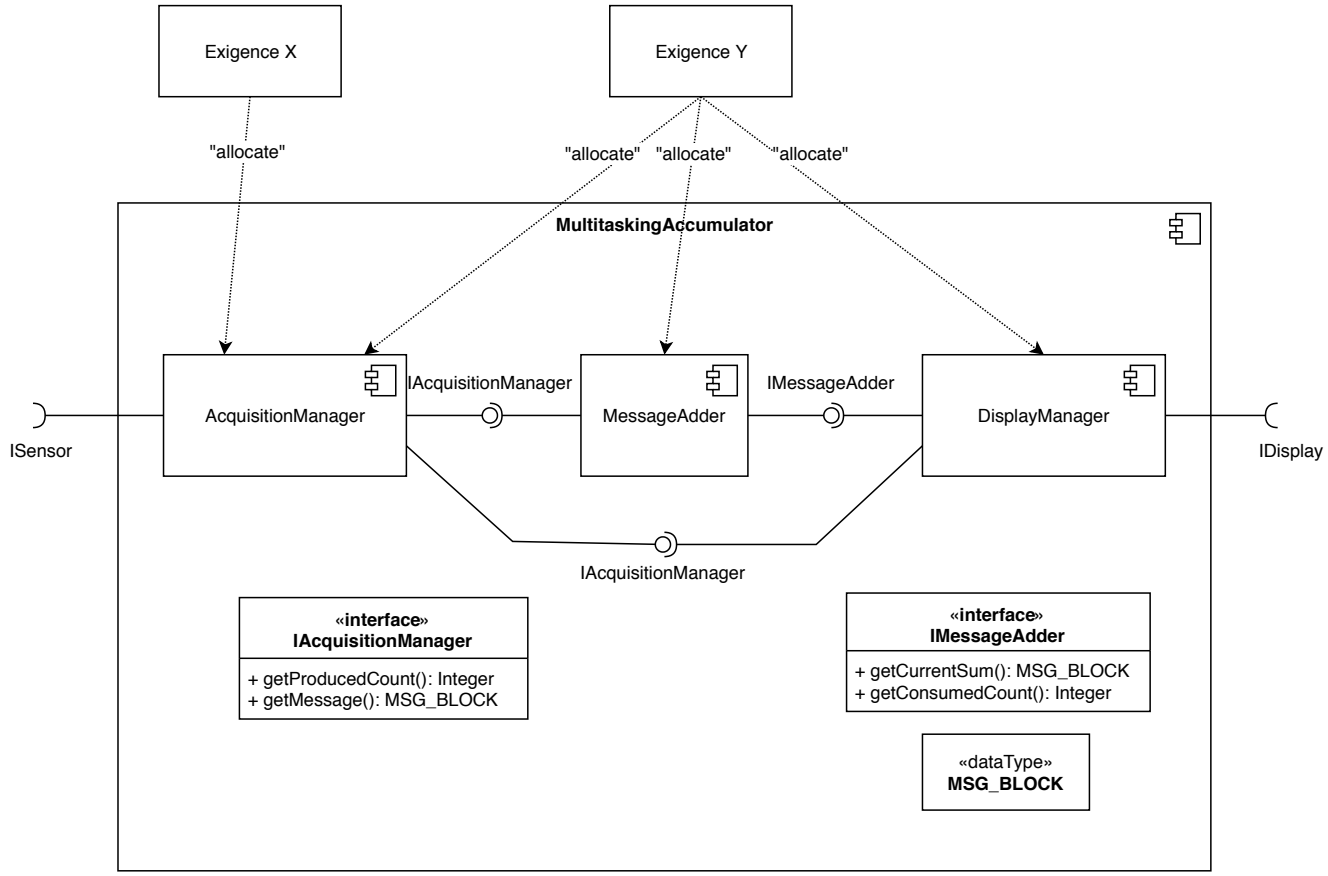

Figure 3: Architecture Logicielle avec allocation des exigences. Un exemple d'allocation d'exigence sur les composants logiciels à modifier/compléter.

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 de type MSG_BLOCK. C'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. La fonction `print` permet d'afficher le nombre de messages produits, consommés et la différence entre les deux.
- L'utilisation du module `msg.h`/`msg.c` qui fournit des fonctions utilitaires pour sommer deux messages et vérifier le checksum d'un message. Le checksum d'un message est calculé grâce à ou logique bit à bit sur chaque entier stocké dans le message (les 256 entiers d'un 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.

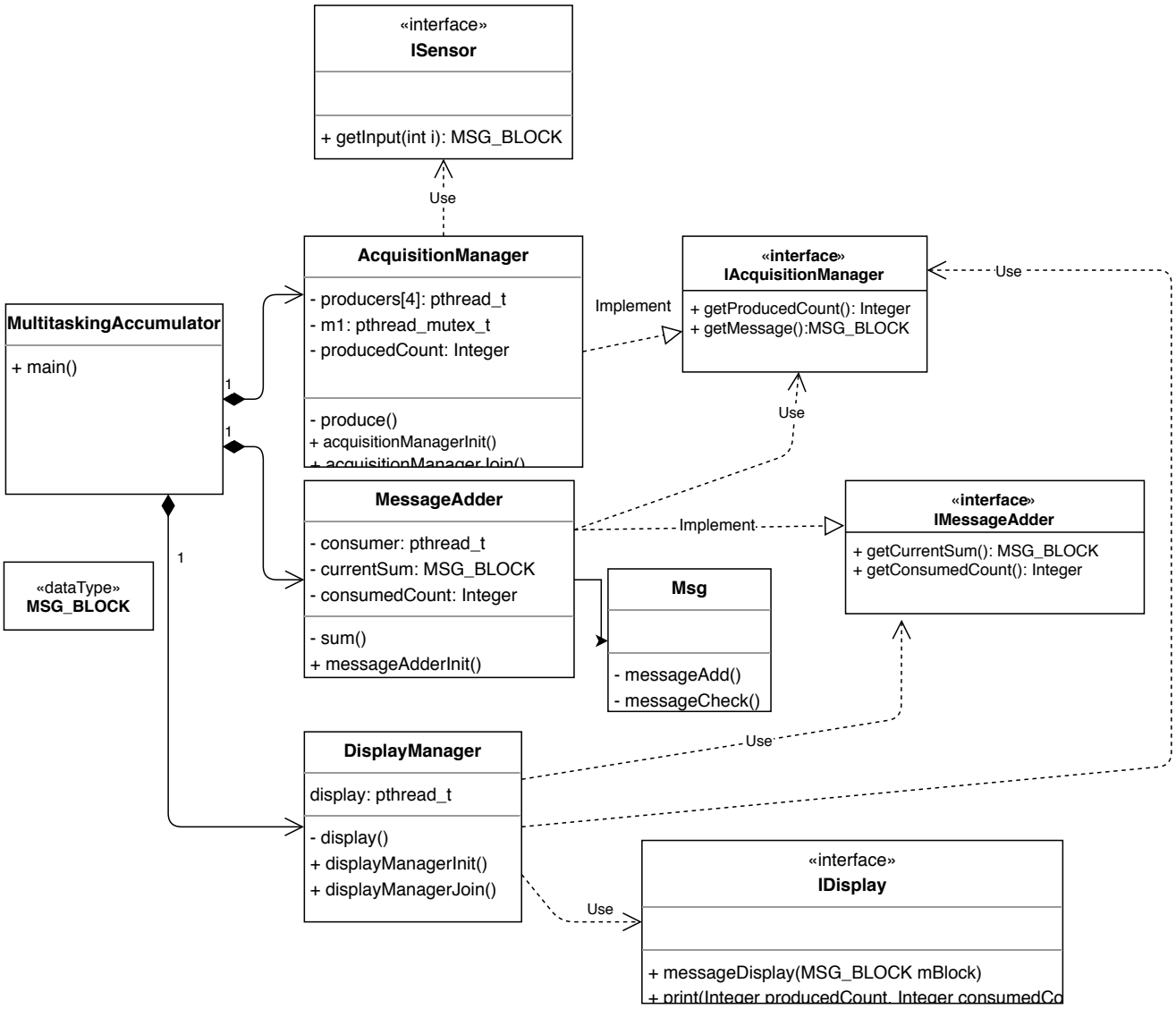

Figure 4: Architecture Logicielle Détaillée de MultitaskingAccumulator. La description de l'architecture logicielle détaillée mise en oeuvre.

3. A quelle stratégie de sûreté la méthode messageCheck répond-t-elle ?
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. Un exemple de diagramme de séquence est illustré en figure 5. Vous pouvez repartir de cet exemple. Vous devrez mettre à jour le diagramme de classes pour y ajouter les APIs dont vous avez besoin et que vous utilisez dans le diagramme de séquence.

   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 (voir `getMessage` dans la figure 5). 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.

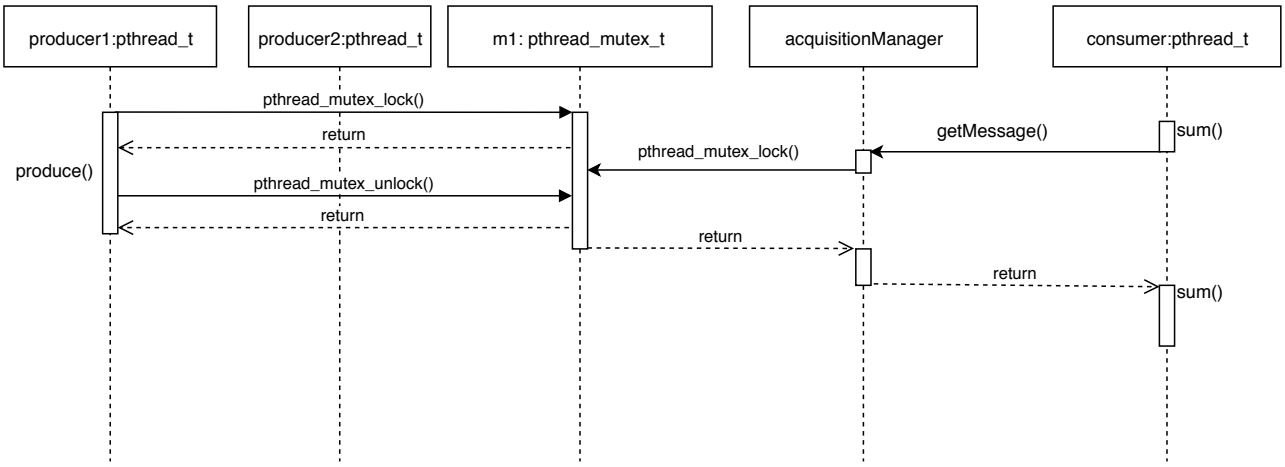

Figure 5: Diagramme de séquence exemple pour illustrer le comportement dynamique de MultitaskingAccumulator. Un exemple de description du comportement dynamique à compléter/modifier.

Implémentation dirigée par les événements

6. Implémentez votre conception (Implémentation + Exécution) et montrez un résultat d'exécution.