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. Identifiez les processus que vous allez mettre en jeu pour exécuter votre programme.
10. Nous allons utiliser le logiciel plantuml pour réaliser de manière automatique nos diagrammes de séquences UML (L'objectif de ce TP n'est pas de vous apprendre l'UML). Un éditeur plantuml est utilisable directement en ligne plantText. Copiez/collez le code présent dans le fichier `diagrams/preambule.puml` dans l'éditeur en ligne puis cliquez sur `Refresh`. Cela doit vous donner le diagramme ci-dessous. Complétez le diagramme ci-dessous expliquant la conception détaillée de ce processus et de ce thread en remplaçant les `[TODO]`. Vous devez exporter votre diagramme pour l'intégrer à votre compte rendu. Seul le diagramme dans le compte rendu compte.
11. En partant de l'hypothèse qu'il n'existe pas d'exigences temporelles sur ce programme, sa conception est-elle complète ?

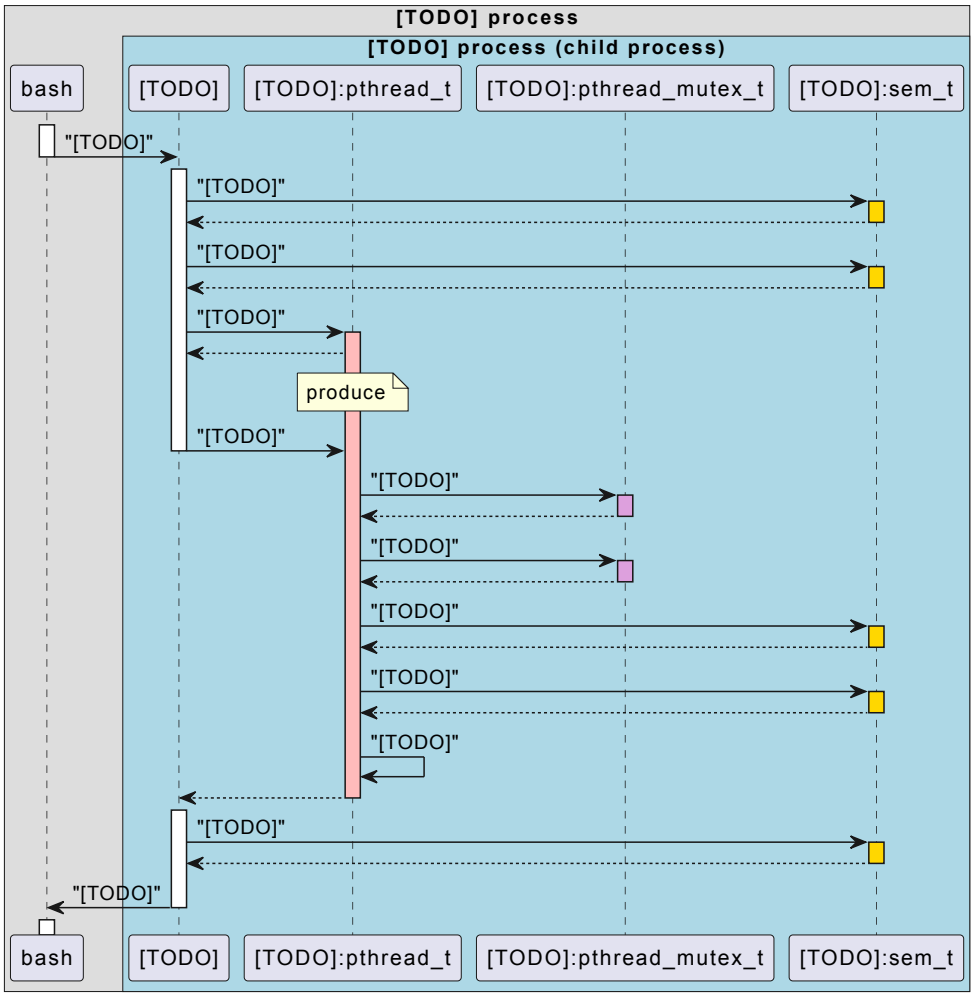

Figure 1: Conception détaillée du processus de découverte des APIs POSIX. La description de l'architecture dynamique du processus de découverte des APIs POSIX.

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.

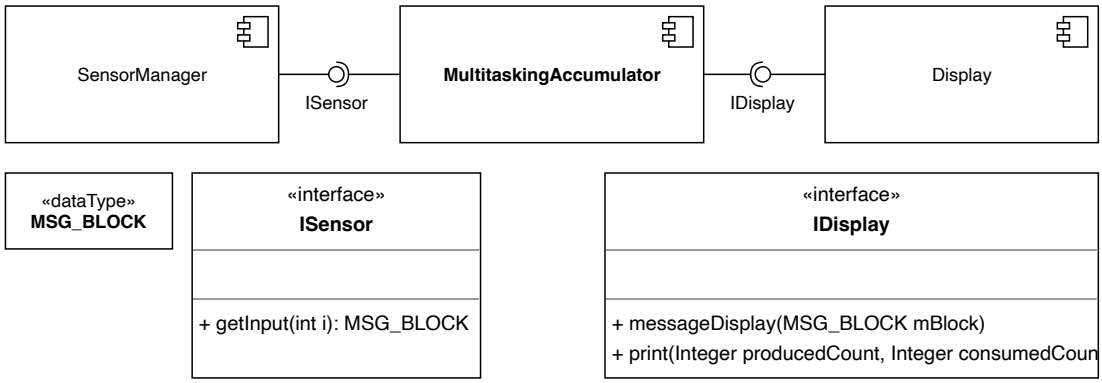

Figure 1: Architecture Système. La description de l'architecture système dans laquelle notre logiciel est intégré.

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

- **Exigence 1** : MultitaskingAccumulator doit produire sur sa sortie `IDisplay` (`messageDisplay`) le cumul de quatre `ISensor` dès qu'une entrée est acquise sur `ISensor`. 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 fournies sur son entrée `ISensor`.
- **Exigence 3** : MultitaskingAccumulator doit garantir que les données d'entrée de `ISensor` sont correctement formées avant de les sommer pour produire le cumul sur `IDisplay`, sinon il doit supprimer la donnée acquise et relever un message d'erreur dans ses logs produits sur sa sortie standard.
- **Exigence 4** : MultitaskingAccumulator doit sommer et produire une sortie sur `IDisplay` au plus vite, c'est-à-dire dès qu'une entrée est présente sur `ISensor` sans attendre une nouvelle valeur sur chacune des entrées.
- **Exigence 5** : MultitaskingAccumulator doit produire un diagnostic sur sa sortie `IDisplay` (`print`) de manière asynchrone vis à vis du cumul. Ce diagnostic explicite combien d'entrées ont été acquises, combien ont été sommées et combien restent à sommer. Ces métriques doivent être cohérentes avec le cumul produit sur `IDisplay` (`messageDisplay`). Il n'est pas exigé qu'un diagnostic soit généré pour chaque cumul.

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 successives 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 :

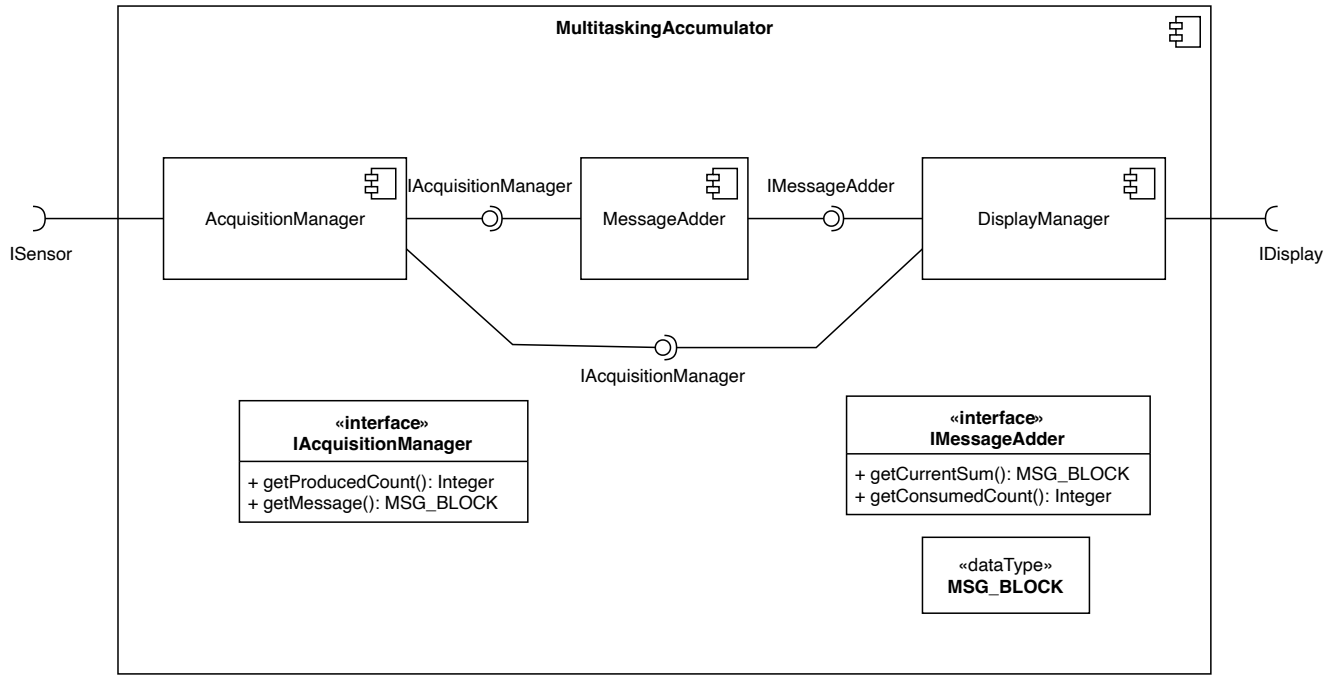

Figure 2: Architecture Logicielle de MultitaskingAccumulator. La description de l'architecture logicielle à mettre en oeuvre.

`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. Pour cela, les sources de ce diagramme à compléter sont rangées dans le fichier `diagrams/conception.drawio` editable avec le programme en ligne draw.io (Importez ce fichier dans l'éditeur en ligne pour gagner du temps). **(0,5 POINT)**

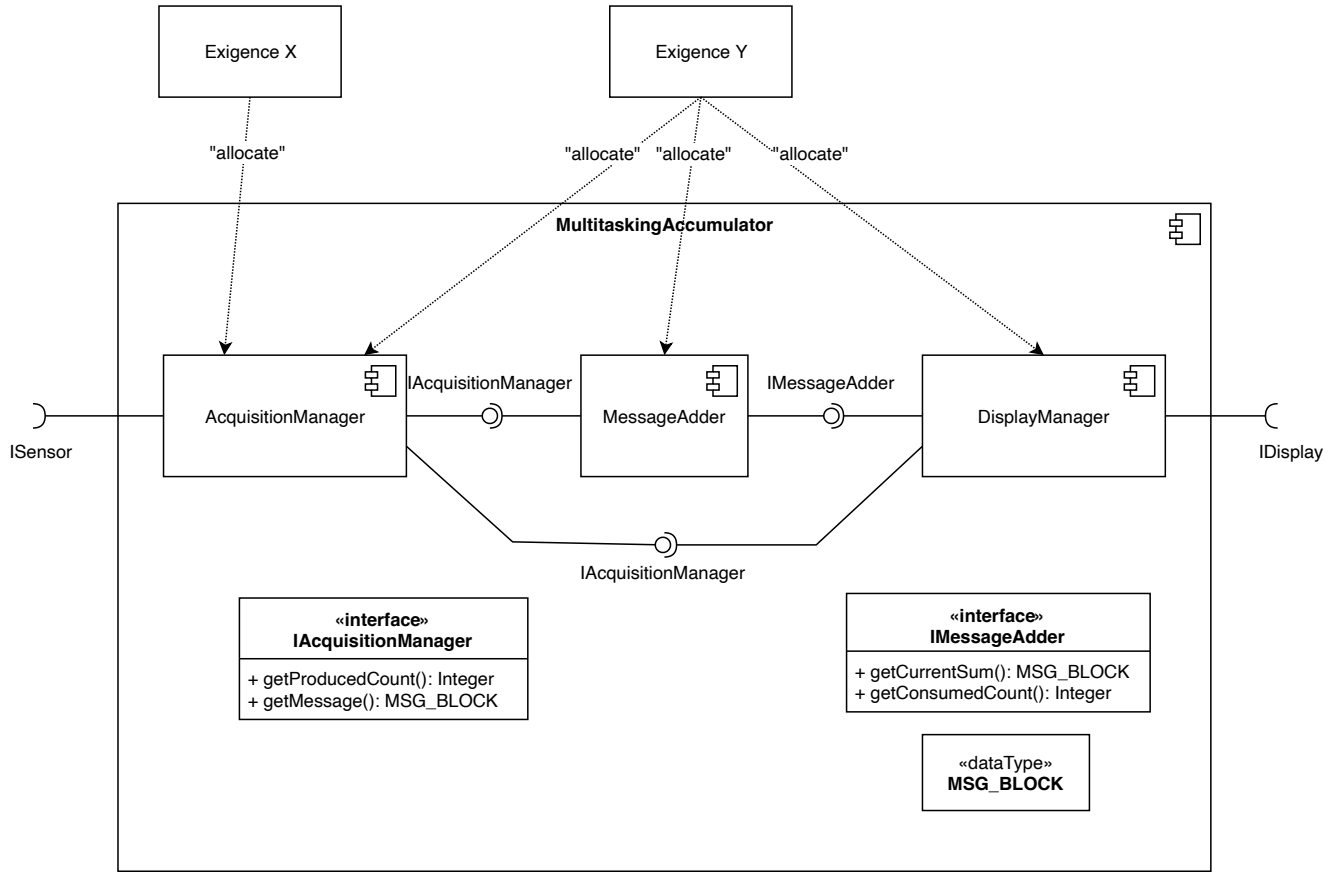

Figure 3: Architecture Logicielle avec allocation des exigences. Un exemple d'allocation d'exigence sur les composants logiciels à modifier/compléter.

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 ? **(1 POINT)**

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.

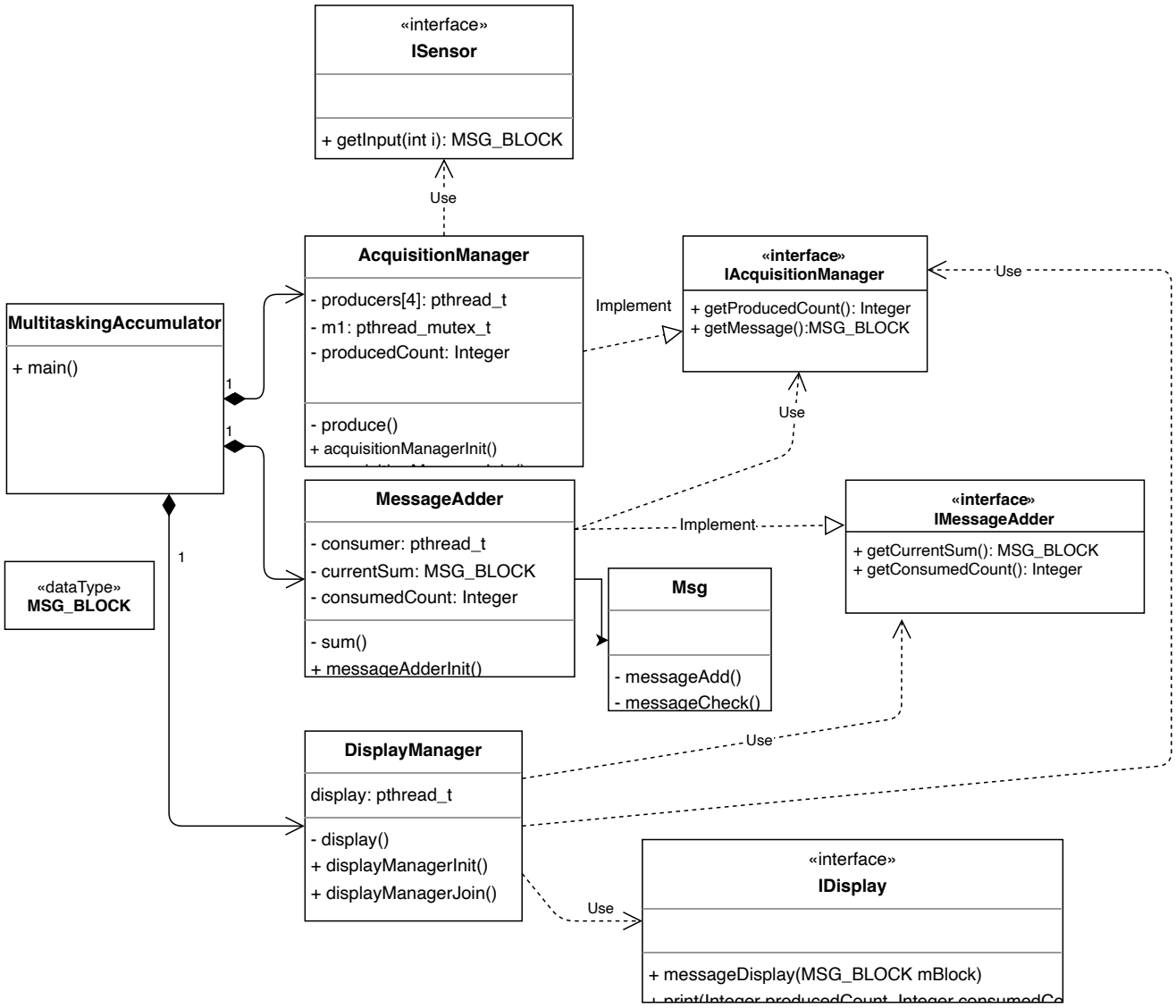

Figure 4: Architecture Logicielle Détaillée de MultitaskingAccumulator. La description de l'architecture logicielle détaillée mise en oeuvre.

3. A quelle stratégie de sûreté la méthode messageCheck répond-t-elle ? **(1 POINT)**
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 ? **(0,5 POINT)**
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. **(3 POINTS)**

   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 et du formalisme `plantuml` que nous avons vu en `préambule`. 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. Vous devrez fournir dans votre rapport ce diagramme de classes et 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.

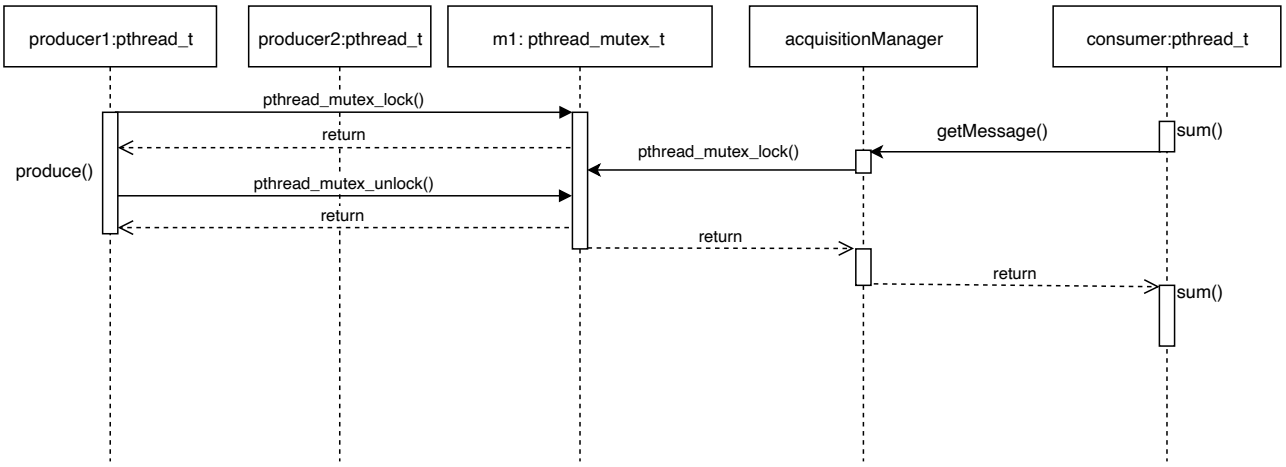

Figure 5: Diagramme de séquence exemple pour illustrer le comportement dynamique de MultitaskingAccumulator. Un exemple de description du comportement dynamique à compléter/modifier.

Implémentation dirigée par les événements

6. Implémentez votre conception (Implémentation + Exécution) et montrez un résultat d'exécution. **(2 POINTS)**
7. Expliquez comment vous avez rendu cohérent la sortie diagnostic `IDisplay` (`print`) et le cumul produit sur `IDisplay` (`messageDisplay`) ? **(1,5 POINT)**
8. Pourquoi certaines variables sont-elles considérées comme des variables C volatiles dans l'implémentation proposée ? **(1 POINT)**

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

9. 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 ? **(1 POINT)**
10. 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 ? **(1 POINT)**
11. Implémentez cette solution dans la méthode `incrementProducerCount` et `getProducerCount` dans le fichier `acquisitionManagerAtomic.c` et montrez un résultat d'exécution. **(1 POINT)**

Pour compiler votre programme, deux règles dans le makefile est proposée :

- `make atomic` : compile le programme incluant le fichier `acquisitionManagerAtomic.c`
- `make runatomic` : compile le programme incluant le fichier `acquisitionManagerAtomic.c`

12. Proposez une autre solution pour que ces méthodes `incrementProducerCount` et `getProducerCount` en vous basant sur la méthode `atomic_compare_exchange_weak` ? **(1 POINT)**
13. Implémentez cette solution dans la méthode `incrementProducerCount` et `getProducerCount` en définissant deux méthodes `pCountLockTake()` et `pCountLockRelease()`. Montrez un résultat d'exécution. **(1 POINT)**

- `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

Nous avons exécuté plusieurs fois chacun des programmes précédents (`MultitaskingAccumulatorPosix` utilisant les mutex POSIX, `MultitaskingAccumulatorAtomi` utilisant une implémentation C11 Atomic et `MultitaskingAccumulatorTestA` utilisant la méthode C11 atomic_compare_exchange_weak). Nous avons mesuré le nombre d'appel à `getProducerCount`, son temps total d'exécution et son temps moyens d'exécution. Les résultats sont présentés ci-dessous.

```
Process               NbCalls TotalExecTime AvgExecTime
MultitaskingAccumulatorPosix[8193]  8 times, 866us total, 108us avg
MultitaskingAccumulatorPosix[8247]  8 times, 199us total, 24us avg
MultitaskingAccumulatorPosix[8215]  8 times, 393us total, 49us avg
MultitaskingAccumulatorPosix[8270]  8 times, 235us total, 29us avg
MultitaskingAccumulatorPosix[8143]  8 times, 163us total, 20us avg
```

```
MultitaskingAccumulatorPosix[8239]  8 times, 646us total, 80us avg
MultitaskingAccumulatorPosix[8262]  8 times, 149us total, 18us avg
MultitaskingAccumulatorPosix[8159]  8 times, 344us total, 43us avg
MultitaskingAccumulatorPosix[8176]  8 times, 498us total, 62us avg
MultitaskingAccumulatorPosix[8278]  8 times, 182us total, 22us avg

MultitaskingAccumulatorAtomi[9092]  8 times, 123us total, 15us avg
MultitaskingAccumulatorAtomi[9116]  8 times, 124us total, 15us avg
MultitaskingAccumulatorAtomi[9100]  8 times, 278us total, 34us avg
MultitaskingAccumulatorAtomi[8905]  8 times, 144us total, 18us avg
MultitaskingAccumulatorAtomi[8819]  8 times, 116us total, 14us avg
MultitaskingAccumulatorAtomi[8897]  8 times, 169us total, 21us avg
MultitaskingAccumulatorAtomi[9083]  8 times, 113us total, 14us avg
MultitaskingAccumulatorAtomi[8889]  8 times, 116us total, 14us avg
MultitaskingAccumulatorAtomi[9108]  8 times, 180us total, 22us avg
MultitaskingAccumulatorAtomi[9067]  8 times, 118us total, 14us avg

MultitaskingAccumulatorTestA[9783]  8 times, 377us total, 47us avg
MultitaskingAccumulatorTestA[9732]  8 times, 234us total, 29us avg
MultitaskingAccumulatorTestA[9757]  8 times, 226us total, 28us avg
MultitaskingAccumulatorTestA[9723]  8 times, 715us total, 89us avg
MultitaskingAccumulatorTestA[9748]  8 times, 131us total, 16us avg
MultitaskingAccumulatorTestA[9773]  8 times, 148us total, 18us avg
MultitaskingAccumulatorTestA[9793]  8 times, 878us total, 109us avg
MultitaskingAccumulatorTestA[9740]  8 times, 635us total, 79us avg
MultitaskingAccumulatorTestA[9801]  8 times, 541us total, 67us avg
MultitaskingAccumulatorTestA[9765]  8 times, 167us total, 20us avg
```

14. Concluez sur les différentes mesures et identifiez une des causes dès que l'on utilise des APIs POSIX **(1 POINT)**

Conception détaillée en utilisant une approche dirigée par le temps

15. Une approche dirigée par le temps est-elle une approche synchrone ou asynchrone ? **(1 POINT)**

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 délai de bout en bout entre l'acquisition d'une entrée et la sortie affichée incluant cette entrée est de maximum 600 ms.

16. En reprenant la représentation par ligne de temps et par messages illustrée dans la figure suivante, proposez une conception dirigée par le temps simplifiée de ce même programme MultitaskingAccumulator. **(1 POINT)**

Une première ligne horizontale sera celle de l'horloge. Les ticks d'horloge seront des points. Ensuite, une ligne par tâche sera représentée. Les points de synchronisation temporels sur ces tâches seront représentés par des lignes verticales.

La représentation des fonctions entre les points de synchronisation temporels seront représentés par des rectangles (il n'est pas nécessaire de détailler le nom des fonctions et leurs nombres). Les messages produits entre les tâches seront représentés par des enveloppes à l'instant ou ces messages sont visibles des consommateurs (voir exemple ci-dessous et celui vu en cours en fin de partie 2).

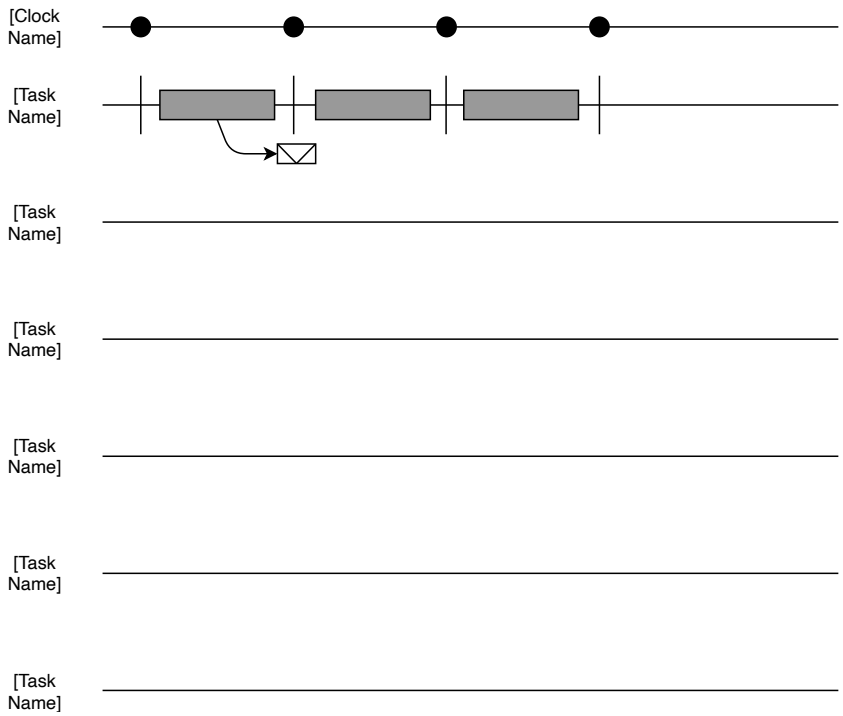

Figure 6: Diagramme de temps de MultitaskingAccumulator dans une approche dirigée par le temps. Description d'une conception dirigée par le temps à compléter/modifier.