UE B0 - Systèmes multitâches

Compte rendu de TP
Systèmes multitâches

THOMAS BOULANGER JULES FARNAULT

19 NOVEMBRE 2024

Table des matières

Introduction

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

1 | Préambule

L'objectif de cette partie est de prendre en main la création d'une tâche et d'un sémaphore en utilisant les APIs POSIX.

On ouvre preambule.c dans Vim. On identifie alors le nom du thread (thread_1), du sémaphore ("/preambule_sem") et du mutex (PTHREAD_MUTEX_INITIALIZER ;) à partir de la ligne 10 :

```
10 #define SEM_NAME "/preambule_sem"
11
12 pthread_t thread_1;
13 sem_t *semaphore;
14
15 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

La création du thread se fait à la ligne 44, son point d'entrée est la fonction *produce* :

```
44 pthread_create(&thread_1, NULL, produce, NULL);
```

L'attente de la fin du thread se fait à la ligne 47 :

```
44 pthread_join(thread_1, NULL);
```

Le thread par défaut du processus courant est la fonction main. Dans notre programme, nous avons deux process : le terminal et le main.

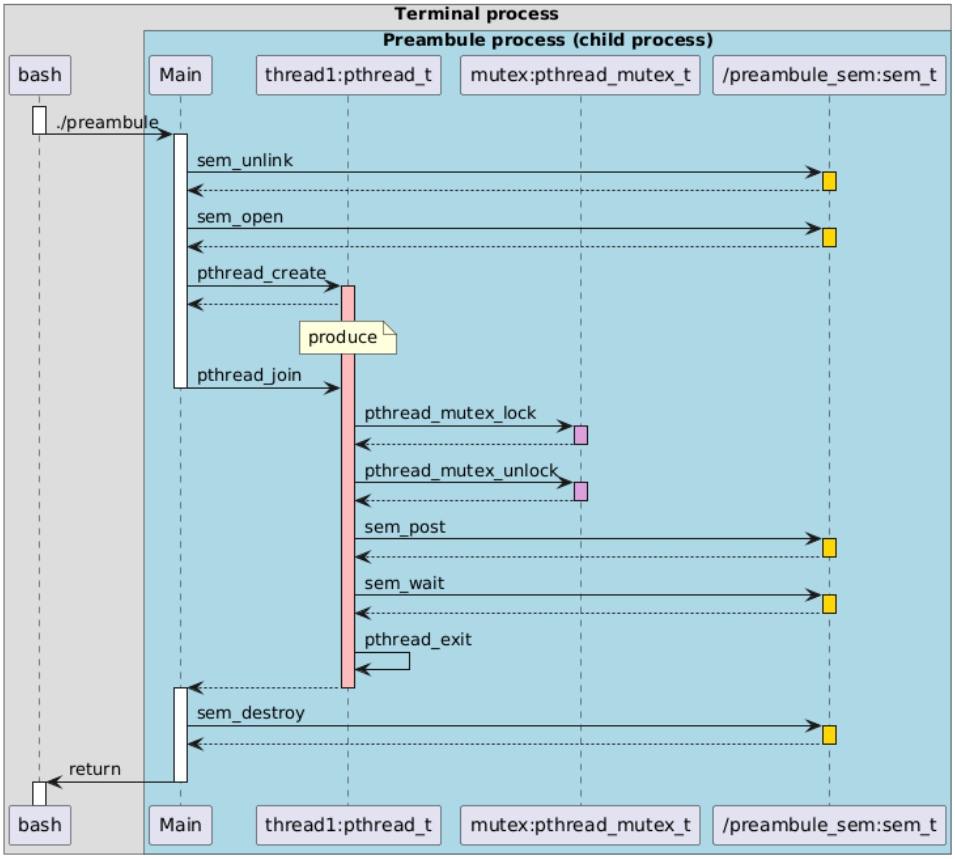

FIGURE 1.1 – Diagramme UML de l'exécution de *preambule.c*

Même si l'on considère qu'il n'y a pas d'exigences temporelles, la conception de ce programme n'est pas complète, car seul le cas nominal est représenté, il manque les cas d'erreurs (erreur de création de sémaphore, de mutex, etc).

2 Premier exercice (Exercice Principal)

Le but de cette partie est de développer un accumulateur (MultitaskingAccumulator) qui réalise l'acquisition de quatre entrées numériques asynchrones auprès d'un composant logiciel externe (SensorManager) et qui somme ces entrées. Les exigences sont définies dans l'énoncé.

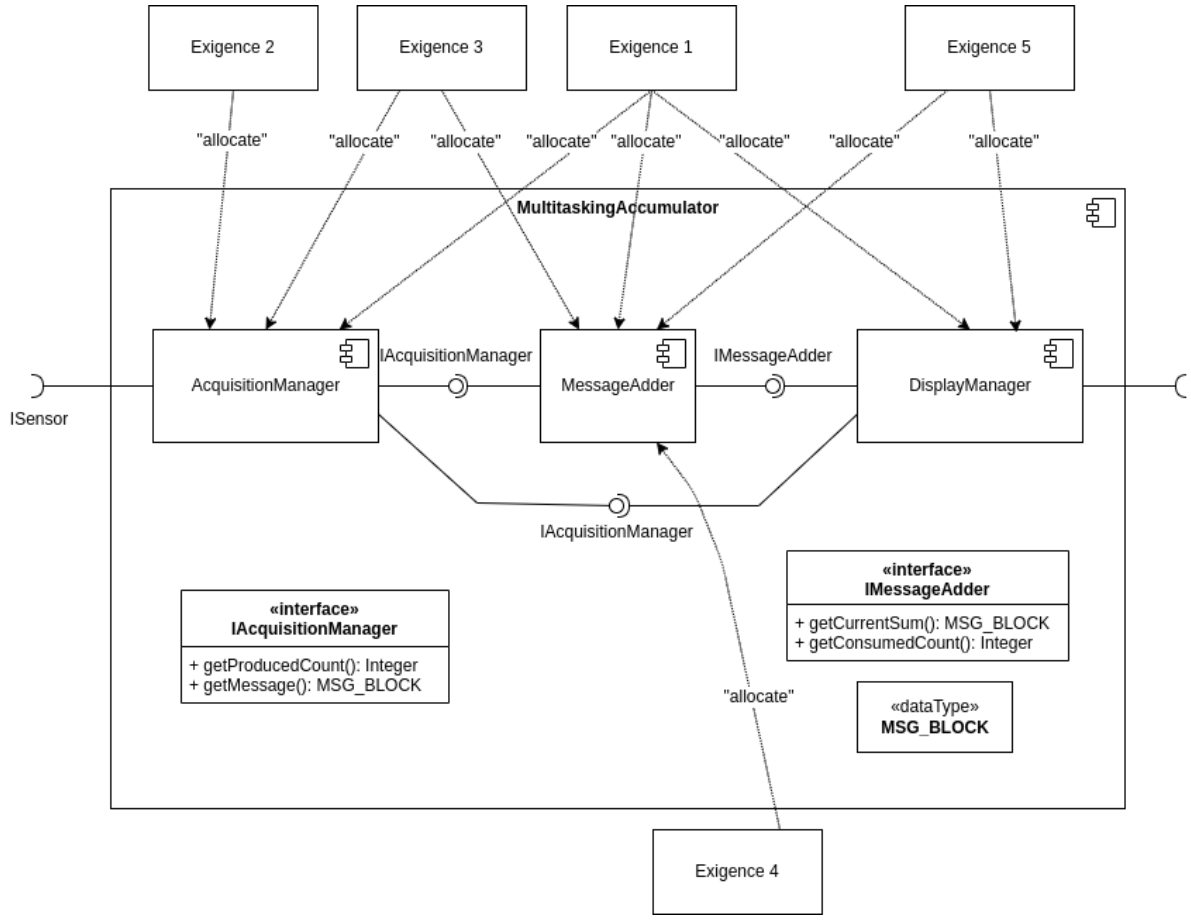

FIGURE 2.1 – Allocation des exigences du système

2.1 - Conception détaillée en utilisant une approche dirigée par les événements

Une approche dirigée par les événements est une approche asynchrone, on agit seulement quand un événement apparaît et ils n'apparaissent pas nécessairement de façon périodique.

La méthode messageCheck vérifie la checksum du message, ce contrôle répond à une stratégie de *Sanity Checking* appartenant à la catégorie des *Failure Detection* et permet de s'assurer qu'il n'y a pas de corruption du message.

Le programme est composé d'un unique process généré dans le main de MultitaskingAccumulator.c. Ce programme contient 7 threads :

- Le thread du main
- 4 threads pour l'Acquisition manager
- 1 thread pour le MessageAdder
- 1 thread pour le DisplayManager

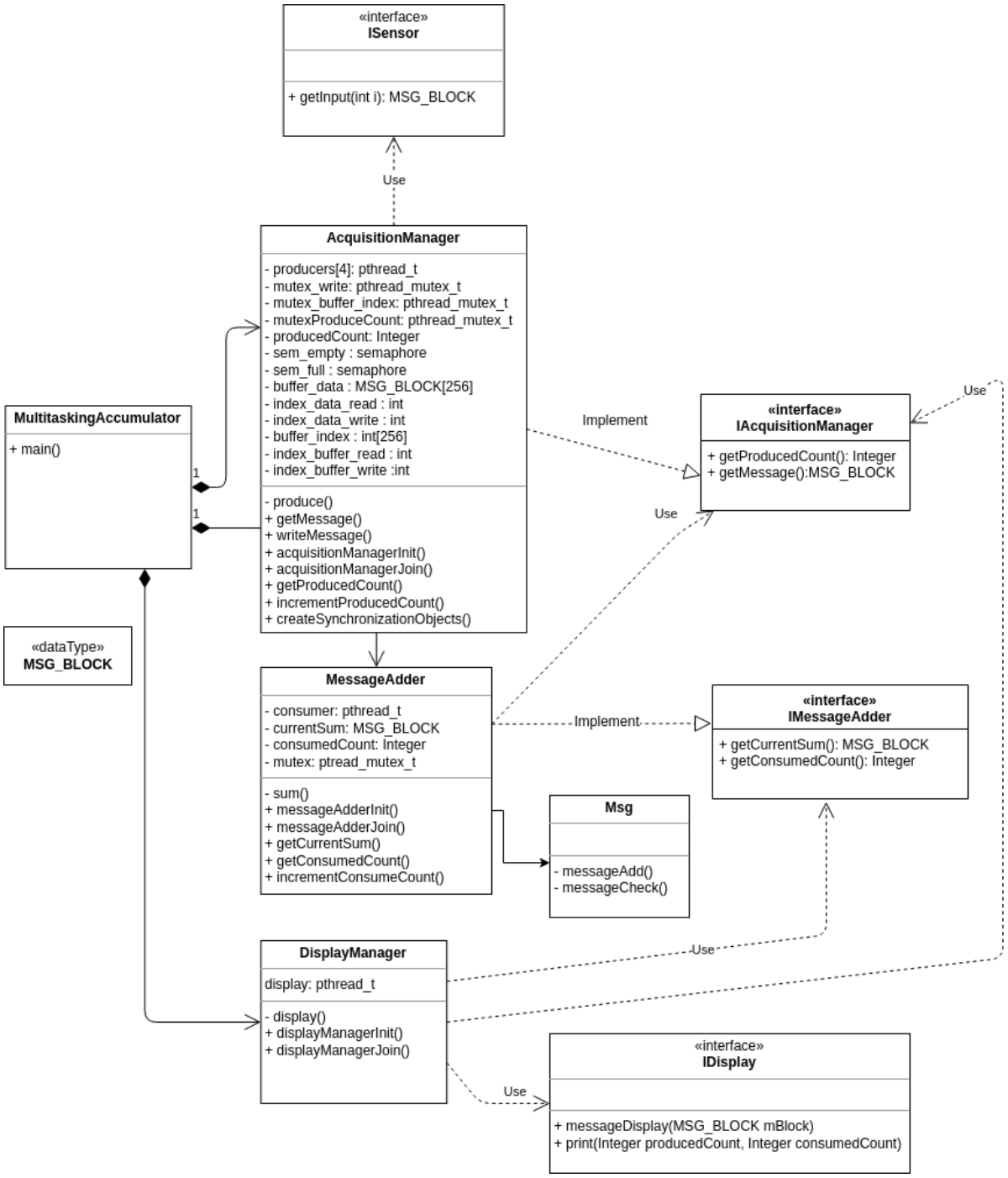

FIGURE 2.2 – Architecture logicielle détaillée de la conception

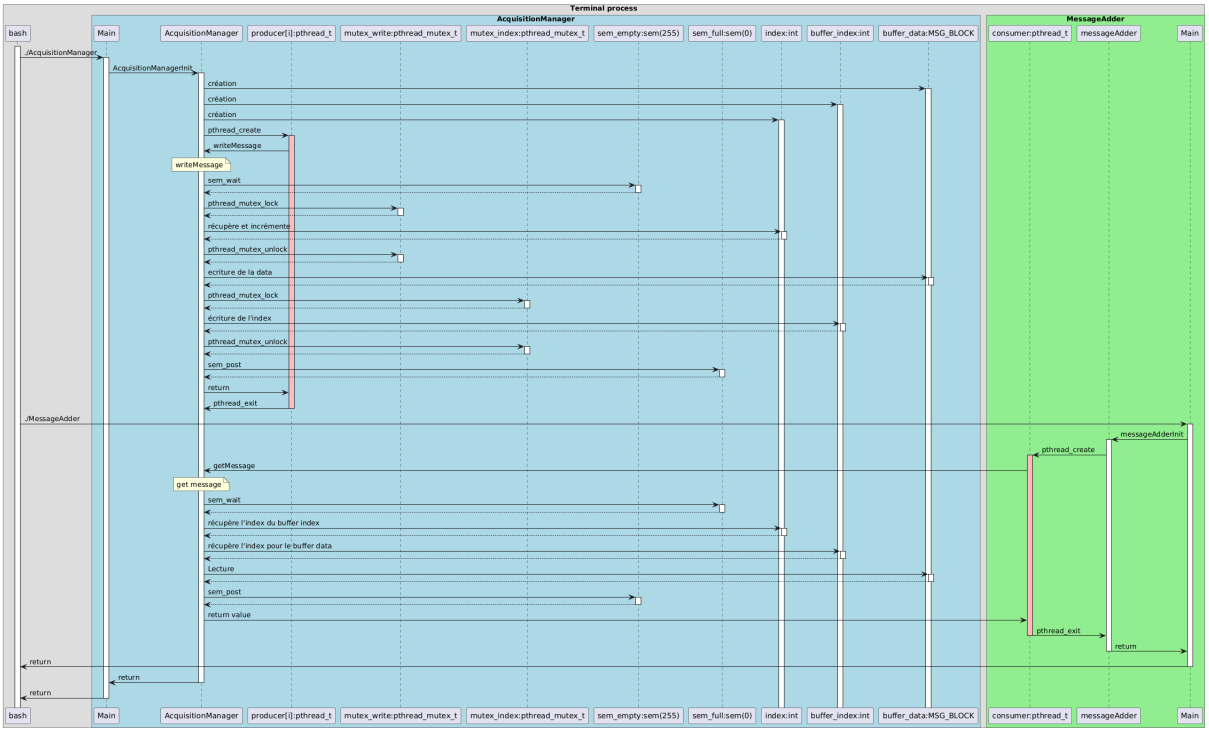

FIGURE 2.3 – Diagramme de séquence entre l'Acquisition manager et le MessageAdder

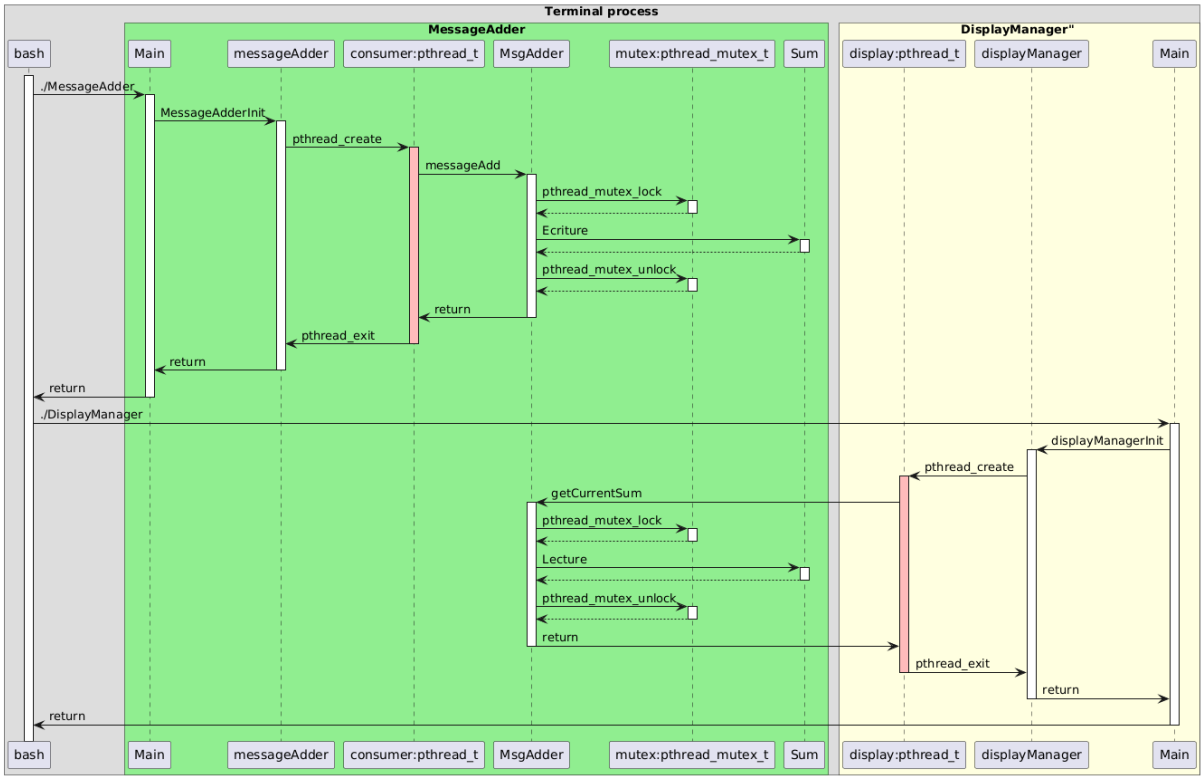

FIGURE 2.4 – Diagramme de séquence entre le MessageAdder et DisplayManager

2.2 - Implémentation dirigée par les événements

On implémente la conception précédente et on regarde sa bonne exécution :

```
cc -pthread -o multitaskingAccumulatorPosix acquisitionManagerPOSIX.o
 ↪  display.o displayManager.o messageAdder.o msg.o
 ↪  multitaskingAccumulator.o sensorManager.o
./multitaskingAccumulatorPosix
[multitaskingAccumulator]Software initialization in progress...
[acquisitionManager]Synchronization initialization in progress...
[acquisitionManager]Synchronization initialization done.
[multitaskingAccumulator]Task initialization done.
[multitaskingAccumulator]Scheduling in progress...
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 3, Consumed messages: 1, Messages
 ↪  left: 2
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 44019 termination
[OK      ] Checksum validated
[acquisitionManager] 44018 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 44017 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 7, Consumed messages: 2, Messages
 ↪  left: 5
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 44016 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 4, Messages
 ↪  left: 4
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
```

```
Message
[displayManager]Produced messages: 8, Consumed messages: 5, Messages left: 3
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 7, Messages left: 1
[OK      ] Checksum validated
[messageAdder] 44020 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages left: 0
[OK      ] Checksum validated
[OK      ] Checksum validated
```

La cohérence sur IDisplay est garanti par la création d'une nouvelle structure qui contient le compteur ConsumedCount et la somme. Ainsi, on obtient les deux variables en même temps en appelant la fonction getCurrentSum(), ce qui garanti la synchronisation entre les deux variables.

```
typedef struct MSG_BLOCK_TAG2
{
        MSG_BLOCK mBlock;
        unsigned int consumedCount;
} MSG_BLOCK_with_ConsumedCount;
```

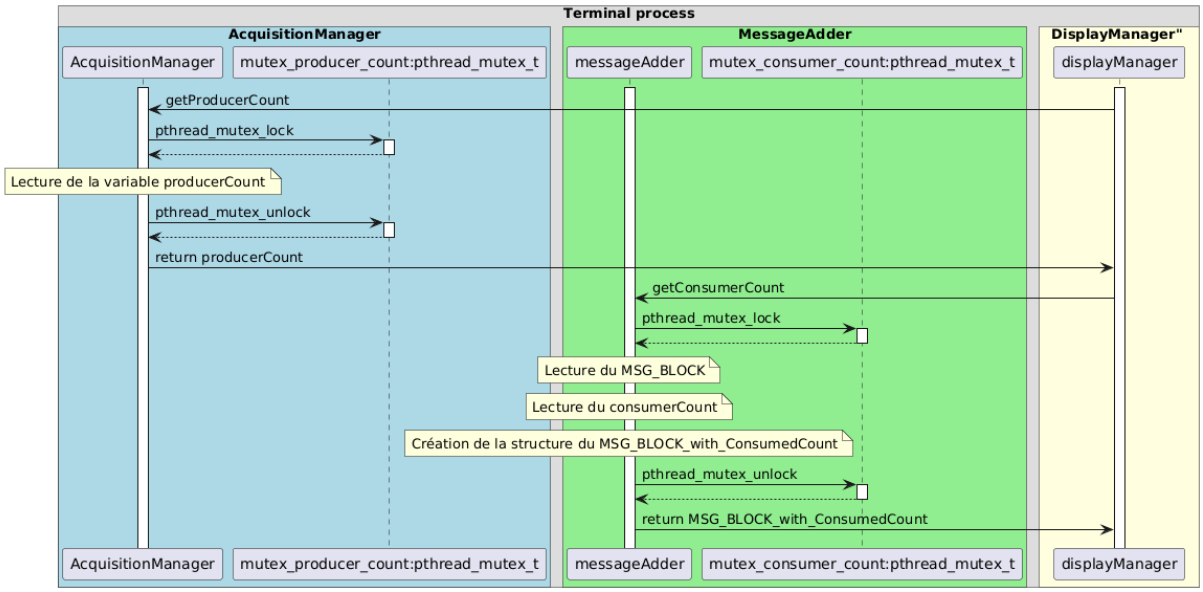

FIGURE 2.5 – Diagramme de séquence pour la cohérence d'information de l'exigence 5

Certaines variables sont considérées comme des variables C volatiles afin de désactiver les optimisations du compilateur sur les ressources partagé entre les threads, ce qui permet de garantir la cohérence spatiale des données.

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

Les processus POSIX possèdent un partitionnement spatial qui permet d'ajouter une sécurité évitant la corruption des données. Cependant, entre les processus, il n'est pas possible d'utiliser des mutexs, il faut également utiliser de la mémoire partagée ou des pipes afin de synchroniser les données. La conception faite précédemment ne sera donc pas utilisable tel qu'elle, il faudra fortement la modifier pour qu'elle puisse répondre aux exigences avec des processus POSIX.

Afin de protéger le compteur permettant compter le nombre de messages produits de manière efficace entre les tâches et sans utiliser des apis POSIX, on peut utiliser le mot clé _atomic en C.

On peut alors retirer le mutex `mutexProduceCount` dédié à la manipulation de `produceCount` :

```
1  _Atomic volatile unsigned int produceCount = 0;
2
3  static void incrementProducedCount(void)
4  {
5          produceCount++;
6  }
7
```

```
unsigned int getProducedCount(void)
{
        unsigned int p = 0;
        p = produceCount;
        return p;
}
```

On lance alors le programme avec cette modification, on a le résultat suivant :

```
./multitaskingAccumulatorAtomic
[multitaskingAccumulator]Software initialization in progress...
[acquisitionManager]Synchronization initialization in progress...
[acquisitionManager]Synchronization initialization done.
[multitaskingAccumulator]Task initialization done.
[multitaskingAccumulator]Scheduling in progress...
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 3, Consumed messages: 1, Messages
 ↪ left: 2
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 92686 termination
[OK      ] Checksum validated
[acquisitionManager] 92687 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 92685 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 7, Consumed messages: 2, Messages
 ↪ left: 5
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 92684 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 4, Messages
 ↪ left: 4
[OK      ] Checksum validated
```

```
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 5, Messages
 ↪  left: 3
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 7, Messages
 ↪  left: 1
[OK      ] Checksum validated
[messageAdder] 92688 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages
 ↪  left: 0
[OK      ] Checksum validated
[OK      ] Checksum validated
```

On va maintenant utiliser une approche Test and Set sur la variable ProduceCount. Pour cela, on utilise la fonction atomic_compare_exchange_weak qui permet de comparer une valeur attendue et de la changer si elles correspondent.

```
_Atomic volatile unsigned int produceCount = 0;
_Atomic volatile int lock = 0;

static void incrementProducedCount(void)
{
        // produceCount++;
        int expected = 0;
        while (!atomic_compare_exchange(&lock, &expected, 1))
        {
        }
        produceCount++;
        lock = 0;
}

unsigned int getProducedCount(void)
{
        unsigned int p = 0;
        int expected = 0;
        while (!atomic_compare_exchange(&lock, &expected, 1))
        {
        }
```

```
        p = produceCount;
        lock = 0;
        return p;
}
```

On exécute la commande `make runtestandset` et on observe le comportement du programme :

```
./multitaskingAccumulatorTestAndSet
[multitaskingAccumulator]Software initialization in progress...
[acquisitionManager]Synchronization initialization in progress...
[acquisitionManager]Synchronization initialization done.
[multitaskingAccumulator]Task initialization done.
[multitaskingAccumulator]Scheduling in progress...
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 3, Consumed messages: 1, Messages
 ↪ left: 2
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 120571 termination
[OK      ] Checksum validated
[acquisitionManager] 120570 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 120569 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 7, Consumed messages: 2, Messages
 ↪ left: 5
[OK      ] Checksum validated
[OK      ] Checksum validated
[acquisitionManager] 120568 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 4, Messages
 ↪ left: 4
[OK      ] Checksum validated
[OK      ] Checksum validated
```

```
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 5, Messages
 ↪  left: 3
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 7, Messages
 ↪  left: 1
[OK      ] Checksum validated
[messageAdder] 120572 termination
[OK      ] Checksum validated
[OK      ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages
 ↪  left: 0
[OK      ] Checksum validated
[OK      ] Checksum validated
```

Les résultats présentés dans le sujet montre que le programme Atomic est le plus rapide par rapport à Posix et TestAndSet. Cela vient du fait que le programme atomique ne fait pas d'appel système (syscall) et donc de changement de contexte, ce qui réduit considérablement le temps d'exécution. La gestion de l'atomicité est gérée directement par le compilateur.

2.4 - Conception détaillée en utilisant une approche dirigée par le temps

Une approche dirigée par le temps est une approche synchrone. C'est-à-dire que l'on a une garantie temporelle sur la disponibilité des données

Conclusion

Ce TP nous a permis de découvrir de manière pratique le développement et la prise en compte de la sécurité des systèmes multitâche à travers diverse exemples (POSIX, Atomic, Test and set) en utilisant des diagrammes UML et des représentations de l'architecture logicielle.