TP - Systèmes multitâches

COELHO RUBACK Arthur
SANTOS SANO Matheus

CEA, France

Novembre 2023

Préambule

En lignes générales le programme contient une fonction "produce" qui est prête à être exécuté pour une thread d'autre que main. Les indices pour cela sont notamment le type de retour, "void*", et la fin d'exécution qui, au lieu de finir avec "return", finit avec "pthread_exit()". Sur les déclarations initiaux, on observe la déclaration d'un objet thread appelé "thread_1", la déclaration d'un objet sémaphore appelé "semaphore" et d'un mutex appelé "mutex". Sur le main du programme il y a respectivement une partie de initialisation du sémaphore et une partie de lancement et attente d'une thread. Pour initialiser le sémaphore le code d'abord réinitialise au sein du système d'exploitation les sémaphores avec ce nom, pour éviter que d'autres problèmes peuvent causer un échec d'exécution. Ensuite le sémaphore est initialisé avec un valeur initial définit, dans ce cas 0. Pour la création d'une thread la commande "pthread_create()" est utilisé avec le nom de la fonction à exécuter comme argument, ainsi que sur quelle objet thread cette thread sera liée et enfin les arguments, dans ce cas NULL. Ensuite le main attends que la thread finisse sa exécution avec la commande "pthread_join". Ce code initialise une autre thread eh dehors que la thread principale du processus (main), la fonction produce. Pour executer ce code il ne faut que le compiler et executer. Cela pourrait être fait manuelment avec GCC ou de façon plus pratique : "make runpreambule".

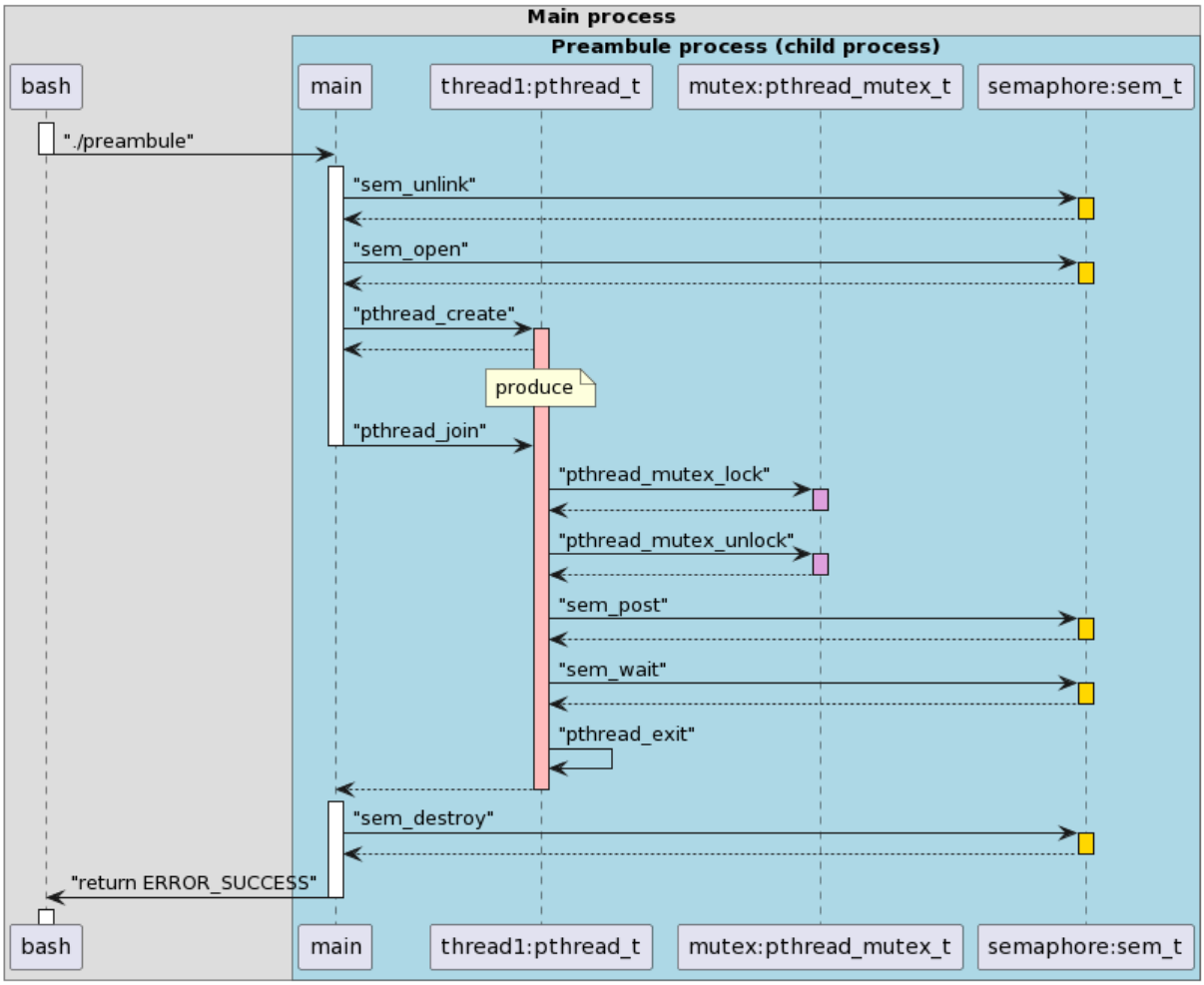

FIGURE 1 – Conception détaillée du processus de découverte des APIs POSIX.

Sur la conception du programme, elle ne peut pas être considère complète parce qu'elle ne montre pas le cas d'erreur, notamment quand il y a un erreur sur "sem_open".

Premier Exercice

1)

L'architecture logicielle avec les exigences attribuées est présentée dans la Figure 2.

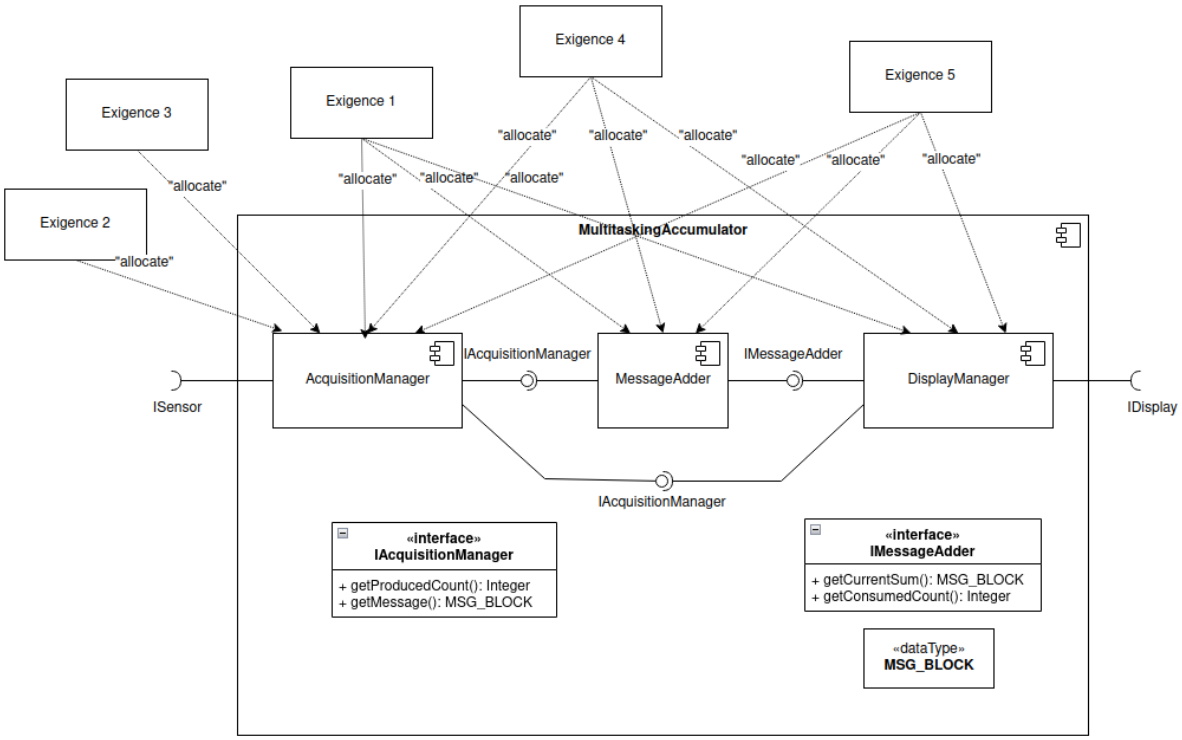

FIGURE 2 – Architecture logicielle avec allocation des exigences.

2)

L'approche dirigée par les évènements est une approche asynchrone. Conformément aux exigences 1 et 4, MultitaskingAccumulator doit sommer et produire une sortie dès qu'une entrée est présente sans attendre une autre entrée. De cette façon, l'approche dirigée par les évènements doivent être une approche asynchrone (Event-Trigger).

3)

La méthode *messageCheck* répond à la stratégie de sûreté de vérification d'intégrité basée sur un checksum, Sanity Checking. Cette méthode garantit que les données n'ont pas été altérées ou corrompues lors de leur transmission ou de leur stockage.

Quand un message est créé, un checksum est calcule en effectuant un XOR sur tous les éléments de *mData* et on stocke le résultat dans "*checksum*" du message. La méthode *messageCheck* recalcule le checksum en effectuant aussi un XOR des éléments de *mData* (*tcheck*) et compare le résultat avec le "*checksum*" stocké dans le message. Si le résultat du calcul du checksum (*tcheck*) correspond au "*checksum*" dans le message, la vérification est réussie , sinon la méthode affiche une erreur de vérification.

4)

Il y a 1 processus (**main()**) et 6 fils d'exécution POSIX supplémentaires (**pthread_t producers[4]**, **pthread_t consumer** et **pthread_t displayThread**).

5)

Pour la nouvelle conception il faut changer l'architecture logicielle de MultitaskingAccumulator, en ajoutant des sémaphores et des mutex. Le diagramme de classes de cette nouvelle architecture est présenté dans la Figure 3.

Par ailleurs, la causalité des traitements et les utilisations des mutex et des sémaphores sont présentés dans le diagramme de séquence illustré dans le fichier joint au rapport.

En bref, il y a 6 threads, 2 sémaphores et 5 mutex. Les threads sont :
— **producers[4]** : pour exécuter la fonction *produce()* ;
— **consumer** : pour exécuter la fonction *sum()* ;
— **displayThread** : pour exécuter la fonction *display()* ;

Le 6 mutex sont utilisés pour empêcher que des ressources partagées soient modifiées et utilisées en même temps. Comme l'architecture étant une FIFO multiRead et multiWrite, les mutex **mutex_ilibre**, **mutex_iplein**, **mutex_jlibre**, **mutex_jplein** sont chargés de protéger les index et d'autoriser plusieurs processus de lecture et/ou d'écriture en même temps. Les mutex **mutex_cons_cnt** et **mutex_prod_cnt**, à leur tour, sont récursifs, c'est-à-dire, ils peuvent être bloqués plusieurs fois par le même processus sans provoquer de deadlock. Dans le *displayThread*, les deux sont bloqués deux fois : pour imprimer le *producedCount* et le *consumedCount*, et pour s'assurer que ces valeurs sont bien les valeurs mises à jour.

Les 2 sémaphores (**sem_nFull** et **sem_nEmpty**) sont chargés de synchroniser les activités du producteur et du consommateur. Le consommateur obtient un message lorsque le sémaphore **sem_nFull** est disponible. Dès l'obtention du message, le consommateur libère le sémaphore **sem_nEmpty**, autorisant la lecture d'un nouveau message. De cette manière, le producteur peut lire un nouveau message lorsque le sémaphore **sem_nEmpty** est disponible. Après lecture, le producteur libère le sémaphore **sem_nFull**, autorisant le consommateur à obtenir le message.

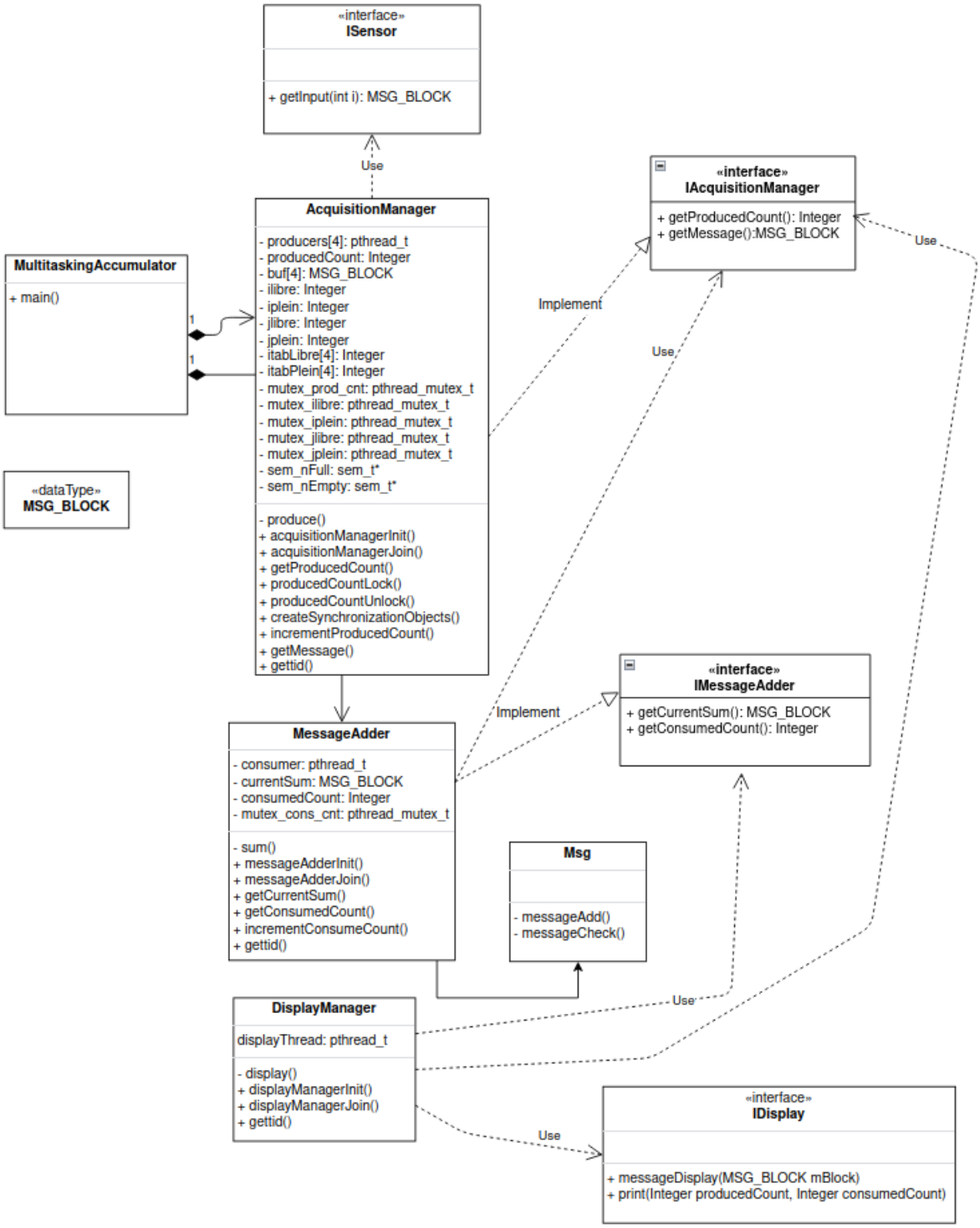

FIGURE 3 – Architecture logicielle détaillée de MultitaskingAccumulator.

6)

```
./multitaskingAccumulatorPosix
[multitaskingAccumulator]Software initialization in progress...
[acquisitionManager]Synchronization initialization in progress...
[acquisitionManager]Semaphores created
[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
Message
[displayManager]Produced messages: 3, Consumed messages: 1, Messages left: 2
[OK       ] Checksum validated
[OK       ] Checksum validated
[acquisitionManager] 31744 termination
[OK       ] Checksum validated
[acquisitionManager] 31745 termination
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 6, Consumed messages: 3, Messages left: 3
[OK       ] Checksum validated
[acquisitionManager] 31743 termination
[OK       ] Checksum validated
[acquisitionManager] 31742 termination
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 4, Messages left: 4
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 6, Messages left: 2
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 7, Messages left: 1
[messageAdder] 31746 termination
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages left: 0
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages left: 0
[displayManager] 31747 termination
[acquisitionManager]Semaphore cleaned
[multitaskingAccumulator]Threads terminated
```

7)

Les variables volatiles sont celles qui peuvent être changés par plus d'une thread et alors le compilateur doit se charger de ne faire pas d'optimisation sur celles ci.

8)

Il y a des différences entre créer une tâche et un processus. La principale est l'étanchéité de la mémoire pour le processus. Le système d'exploitation garantie qu'un processus quelconque ne peux pas accéder au espace mémoire d'un autre processus. Cette caractéristique est très intéressante pour les systèmes orientés sûreté de fonctionnement vu qu'on peut isoler les tâches avec des criticités différentes et par conséquence éviter qu'une tâche défaillante enchaîne des erreurs sur d'autres. Cela est utilisé notamment pour grouper des tâches avec un taux de défaillance similaire. Il est néanmoins problématique dans certains cas vu que le mécanisme de partage "mémoire partagé" entre threads n'est plus possible.

Une autre caractéristique intéressante est que les processus ont leurs propres signaux. Ainsi, une erreur ou une interruption dans un processus POSIX n'affecte pas directement les autres processus. Cependant, une caractéristique qui ne satisfait pas le concept de sûreté de fonctionnement est le coût élevé des ressources, en particulier de la mémoire, dans la création et la gestion des processus, une fois que chaque processus dispose d'un espace mémoire non partagé.

On n'aurait pas pu utiliser les mêmes choix de concepts car les taches partagent une mémoire entre elles. Donc, d'autres mécanismes de partage doivent être utilisés.

9)

Une solution pour protéger de manière efficace entre les taches sans utiliser les API POSIX est d'utiliser des opérations atomiques qui garantissent la cohérence des données partagées par les différents threads. En utilisant l'opération atomique **atomic_fetch_add**, il est possible d'incrémenter le nombre de messages produits.

10)

CODE

11)

La fonction "atomic_compare_exchange_weak" permet de faire un test_and_set d'une variable avec un retour en cas échéant. Elle peut être utilisé pour créer un mutex. La logique est : il faut copier la valeur de la variable nommée mutex et la mettre en "expected". Ensuite si la valeur est 0, le mutex est occupé et il faut qu'on réessaie plus tard. Si la valeur et 1 probablement le mutex est disponible, on doit donc faire l'opération "atomic_compare_exchange_weak" pour essayer de prendre le mutex. Si on y arrive on l'a, sinon il faut revenir u début de la boucle pour réessayer. Pour libérer le muter il faut d'abord vérifier qu'il est bien pris, sinon cela cause un erreur, et ensuite il est libéré avec la même opération.

12)

CODE

```
./multitaskingAccumulatorTestAndSet
[multitaskingAccumulator]Software initialization in progress...
[acquisitionManager]Synchronization initialization in progress...
[acquisitionManager]Semaphores created
[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
Message
[displayManager]Produced messages: 3, Consumed messages: 1, Messages left: 2
[OK       ] Checksum validated
[OK       ] Checksum validated
[acquisitionManager] 67380 termination
[OK       ] Checksum validated
[acquisitionManager] 67379 termination
[OK       ] Checksum validated
[acquisitionManager] 67378 termination
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 7, Consumed messages: 3, Messages left: 4
[OK       ] Checksum validated
[acquisitionManager] 67377 termination
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 4, Messages left: 4
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 6, Messages left: 2
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 7, Messages left: 1
[messageAdder] 67381 termination
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages left: 0
[OK       ] Checksum validated
[OK       ] Checksum validated
Message
[displayManager]Produced messages: 8, Consumed messages: 8, Messages left: 0
[displayManager] 67382 termination
[acquisitionManager]Semaphore cleaned
[multitaskingAccumulator]Threads terminated
```

13)

Selon les donnés on observe que :

— Le temps moyen pour MultitaskingAccumulatorPosix est haut si comparé aux autres.

— Le temps moyen pour MultitaskingAccumulatorAtomi est faible si comparé aux autres.

— Le temps moyen pour MultitaskingAccumulatorTestA est moyen si comparé aux autres.

Ce résultat est consistent avec celui qu'on espérait vu que l'implémentation POSIX dépend du système d'exploitation et donc exige des changements de processus et d'appels SYSCALL. Le résultat de atomi est plus performant parce qu'il utilise des instructions de base "test_and_set" du processeur.

14)

L'approche dirigée par les temps est une approche synchrone, dans laquelle les événements sont exécutés de manière synchrone au moyen d'une horloge ou d'un cycle de temps déterminé (Time-Trigger).

15)

Considérant que toutes les tâches rentrent dans un interval de 100ms, c'est à dire, qui sont capables de s'exécuter en moins de 100ms ; et que Adder est capable de traiter les 4 messages acqueris, on a le diagramme de l'image 4.

Le temps de traitement maximale est de 300ms entre l'acquisition et la sortie.

Deuxième Exercice

1)

Les ressources partagées qui synchronisent le repas des philosophes sont les fourchettes. Comme il y a 5 fourchettes (une pour chaque philosophe), mais chaque philosophe a besoin de deux fourchettes pour manger, seuls deux philosophes peuvent manger en même temps.

2)

En considérant une situation dans laquelle les 5 philosophes sont présents à chaque place de la table, le scénario dans lequel l'interblocage se produit est celui où tous les philosophes prennent la fourchette de gauche de manière séquentielle. De cette manière, tous les philosophes ont la fourchette de gauche et attendent en même temps que le philosophe de droite ait fini de manger. De cette manière, aucun philosophe ne peut manger et lâcher la fourchette de gauche, puisque tout le monde attend que la fourchette de droite soit lâchée.

3)

Une solution pour éviter l'interblocage est d'utiliser le concept de priorité. Le philosophe qui a pris la fourchette de gauche est plus prioritaire que celui qui n'a encore pris aucune fourchette. De cette manière, le philosophe qui a déjà une fourchette prend la fourchette de droite, ce qui permet d'éviter l'interblocage.

Une autre solution consiste à faire en sorte que les philosophes prennent simultanément les fourchettes de droite et de gauche, ce qui résout le problème de l'emboîtement.

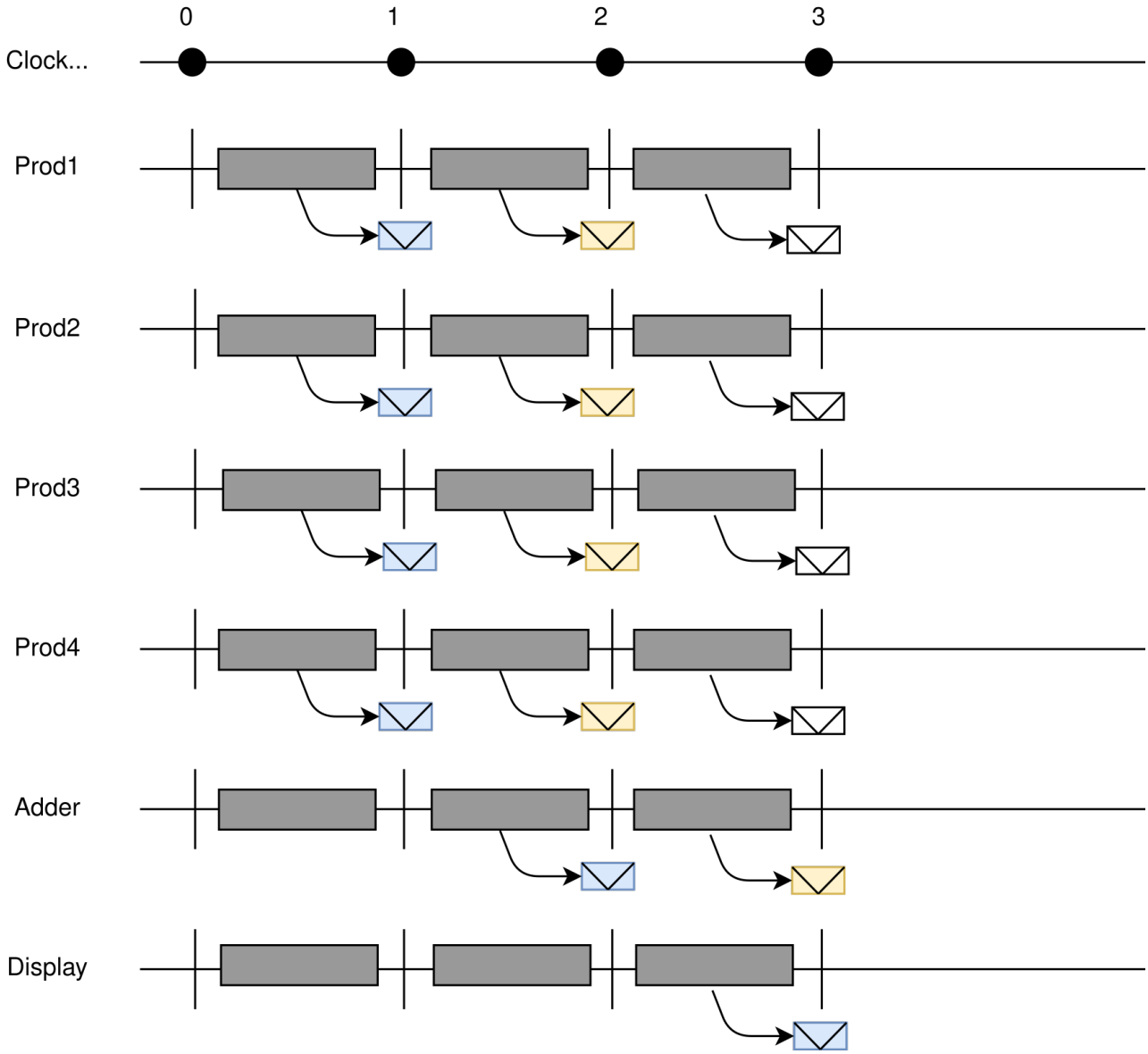

FIGURE 4 – Diagramme de temps de MultitaskingAccumulator dans une approche dirigée par le temps