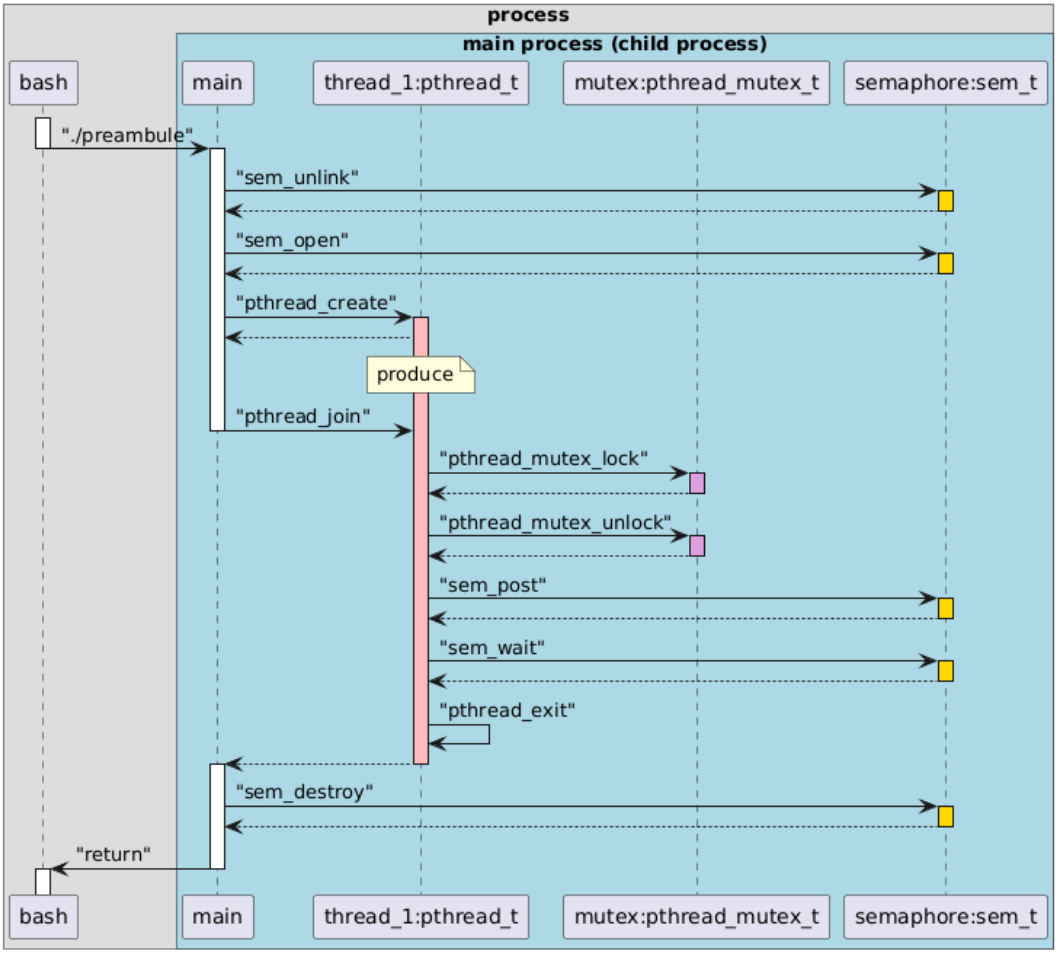
Compte Rendu TP B0

Arnaud FELDMANN - Hugo ALLAIRE

19/11/2024

Préambule

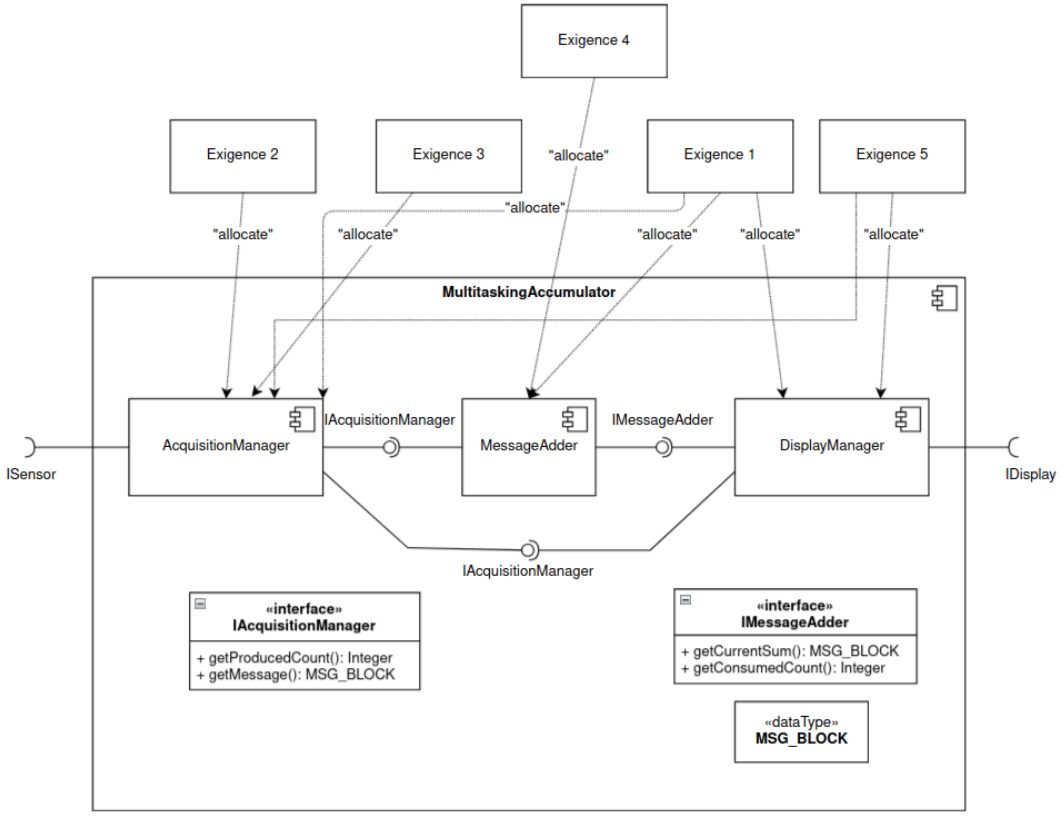
Voici le diagramme de séquence du programme “preambule.c” en utilisant les API POSIX:



Exercice 1

Question 1:

L'exigence 1 a un impact sur les 3 composants. En effet, elle induit que des données *reçues* sont *sommées* et *affichées*.
L'exigence 3 demande que les *données reçues* ne soient transmises à **MessageAdder** lorsqu'elles sont bien formées. Cela ne concerne donc que l'**AcquisitionManager**.
L'exigence 4 demande à ce que l'additionneur n'attende pas l'ensemble de données, c'est donc quelque chose qui concerne le **MessageAdder**.
L'exigence 5 demande à ce que l'**AcquisitionManager** présente une API de diagnostic, à ce que les statistiques de **MessageAdder** soient produites et transmises, et enfin que tout cela soit affiché par le DisplayManager.



Question 2:

La conception dirigée par les évènements est asynchrone car on ne repose ni sur des blocages au sein d'une boucle principale, ni sur des synchronisations basées sur des instants comme dans le modèle time-triggered. On obtient alors quelque chose de naturellement réactif, quand bien la réactivité peut être atteinte dans un modèle time-triggered.

Ici, les 4 threads s'exécutent en même temps si le processeur est multicoeur ou multithread. Ils peuvent aussi être simplement ordonancés dans le cas d'un cœur unique en fonction des blocages. Ces aléas d'ordonnancement existent en réalité *au niveau infrastructure* dans le cas d'une approche time-triggered, mais dans cette dernière approche l'asynchronisme est masqué à l'utilisateur par une couche masquante. L'utilisateur ne remplit que les contraintes à remplir et l'OS se charge du reste.

Les sémaphores et mutex permettent de synchroniser les tâches et protéger les ressources. Cependant, un des désavantages par rapport à une approche time-triggered est que les buffers sont de longueur indéfinie : puisque l'on ne peut pas prévoir l'enchevêtrement des tâches, on ne peut pas connaître le nombre de données maximal produit par les threads producteurs, avant qu'ils ne soient traités par les (ici le) processus consommateur.

Question 3:

La stratégie de sûreté à laquelle répond la méthode messageCheck correspond à la Failure Detection et plus précisément le Sanity Checking. Il s'agit d'un Data Integrity Check.

Question 4:

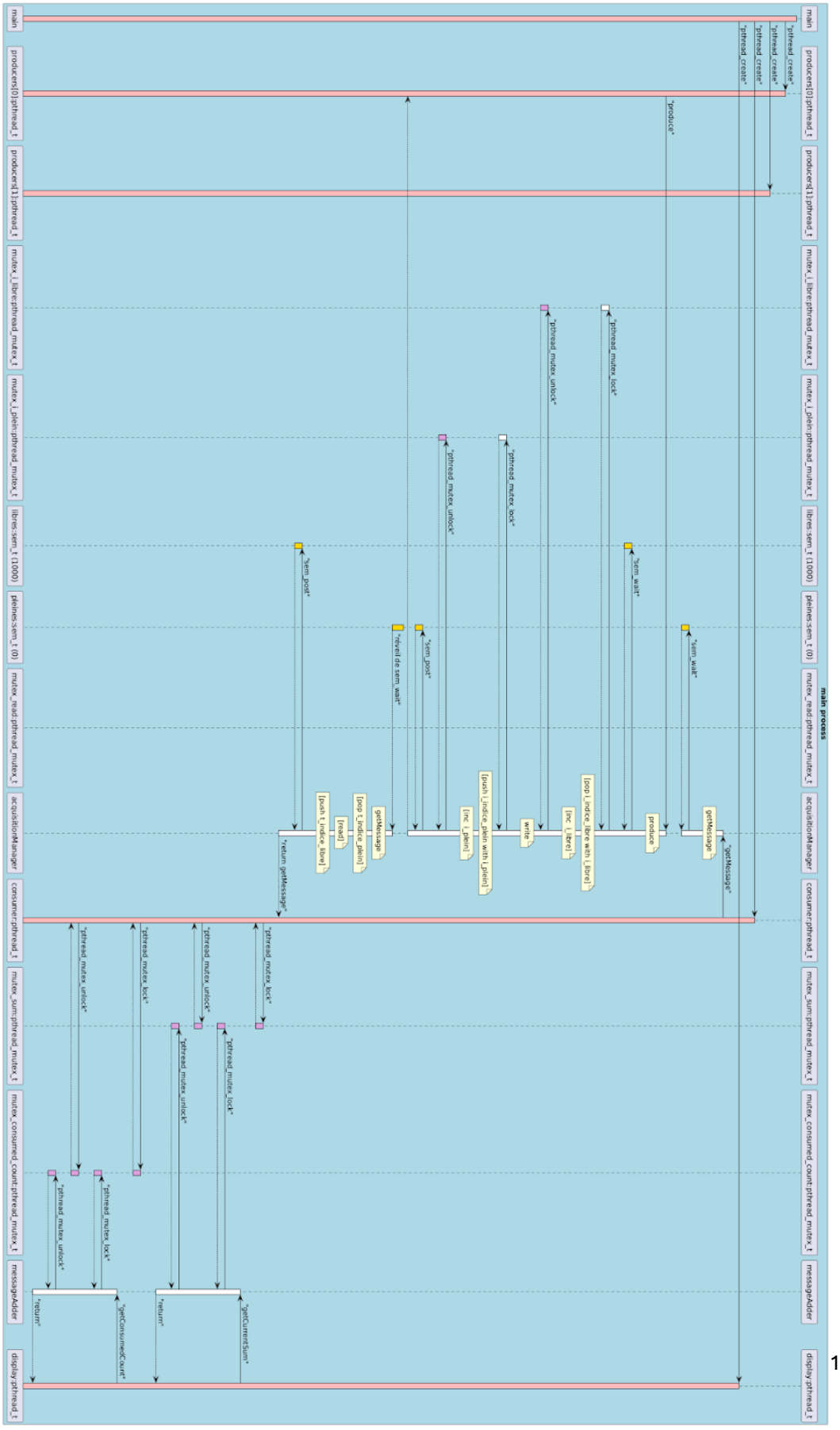
A ce stade de la conception il n'y a qu'un seul processus (celui lancé en exécutant le "main") mais il recouvre plusieurs threads:

- main
- producers * 4
- consumer
- display

Ce qui donne un total de 7 threads.

Question 5:

Voici le diagramme UML de notre implémentation. Les tableaux i_indice_libre et i_indice_plein sont omis, mais décrit après.


main process
main
producers[0]:pthread_t
producers[1]:pthread_t
mutex_libre:pthread_mutex_t
mutex_plein:pthread_mutex_t
libres:sem_t (1000)
pleines:sem_t (0)
mutex_read:pthread_mutex_t
acquisitionManager
consumer:pthread_t
mutex_sum:pthread_mutex_t
mutex_consumed_count:pthread_mutex_t
messageAdder
display:pthread_t
"pthread_create"
"pthread_create"
"pthread_create"
"produce"
"pthread_mutex_lock"
"pthread_mutex_unlock"
"pthread_mutex_lock"
"pthread_mutex_unlock"
"sem_wait"
"sem_post"
"réveil de sem_wait"
"sem_post"
"sem_wait"
[pop i_indice_libre with i_libre]
[inc i_libre]
produce
[push i_indice_plein with i_plein]
[inc i_plein]
write
getMessage
"getMessage"
getMessage
[pop i_indice_plein]
[read]
[push i_indice_libre]
"return getMessage"
"pthread_mutex_lock"
"pthread_mutex_unlock"
"pthread_mutex_lock"
"pthread_mutex_unlock"
"pthread_mutex_lock"
"pthread_mutex_unlock"
"return"
"getConsumedCount"
"return"
"getCurrentSum"

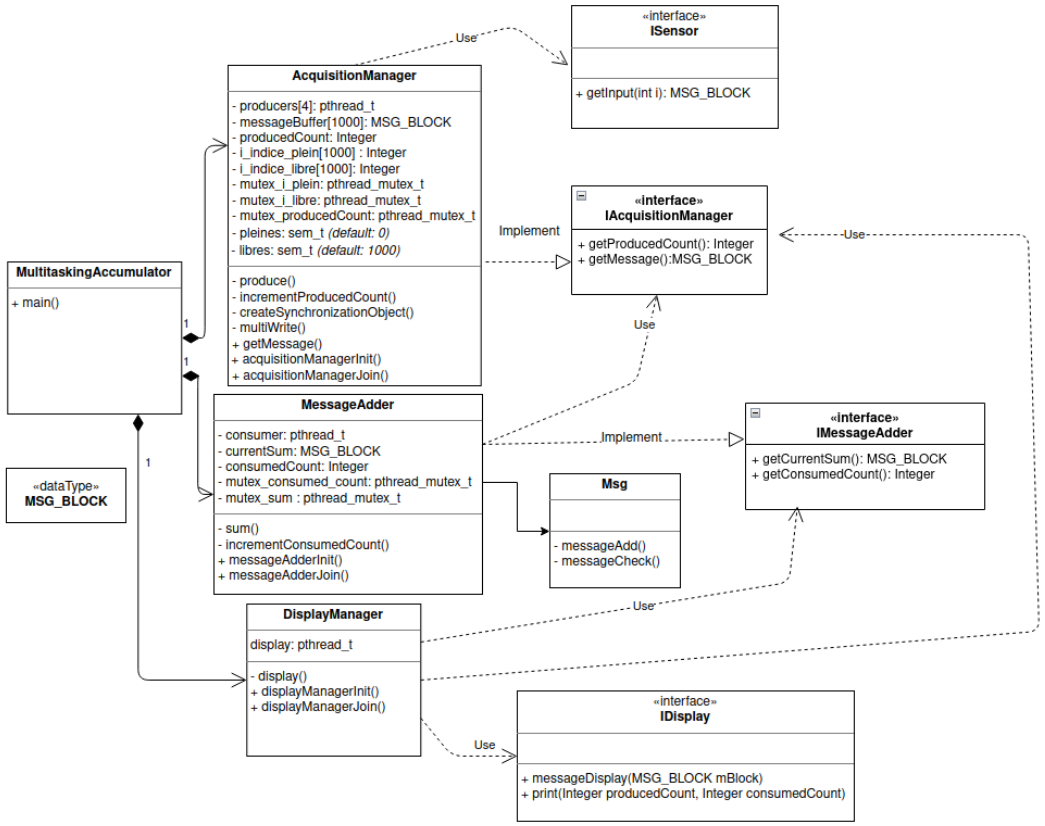
On implémente un multiWrite/singleread dont le pseudo_code est :

```
INIT(libres, 1000)
INIT(pleines, 0)
i_indice_vide initialisé de 0 à 999
i_indice_plein pas besoin d'intialisation car le nombre de cases pleines est vide au début
initialisation des mutex

multiWrite(valeur):
        static int i_libre = 0, i_plein = 0
        int i
        P(libres)
        lock(mutex_i_libre)
        i = i_indice_libre[i_libre]
        i_libre = (i_libre + 1) % N
        unlock(mutex_i_libre)
        write(i, valeur);
        lock(mutex_i_plein)
        i_indice_plein[i_plein] = i
        i_plein = (i_plein + i) % N
        unlock(mutex_i_plein)
        V(pleines)

monoRead(valeur):
        static i_libre =0, i_plein = 0
        int i
        P(pleines)
        i = i_indice_plein[i_plein]
        i_plein = (i_plein + 1) % N
        read(i, valeur)
        i_indice_libre[i_libre]=i
        i_libre = (i_libre + 1) % N
        V(libres)
```

Les i_indice_libre et i_indice_plein permettent aux processus écrivains d'écrire et terminer dans l'ordre qu'ils veulent, et d'occuper le système le moins possible. Les threads peuvent lire les messages de manière concurrente, remplir le buffer à partir des indices qu'ils récupèrent dans i_indice_libre, et terminer dans l'ordre qu'ils veulent vu qu'ils pushent dans l'ordre de terminaison dans i_indice_plein.



Il est donc à noter, puisque messageAdder est le seul processus à lire, que des mutex sur les variables statiques de lecture (sous getMessage) ne sont pas nécessaires.

Question 6:

Pour facilement vérifier le bon fonctionnement du programme, nous avons modifié la fonction getInput pour que la valeur de l'entrée soit égale à l'indice de la valeur dans le message.

Voici les valeurs de sortie après la récupération d'un message:

[displayManager]Produced messages: 2, Consumed messages: 1, Messages left: 1
[OK] Checksum validated
[acquisitionManager] 140211042637568 termination
[messageDisplay]Message:
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200

201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255]

Après 8 messages, voici les valeurs de sortie:

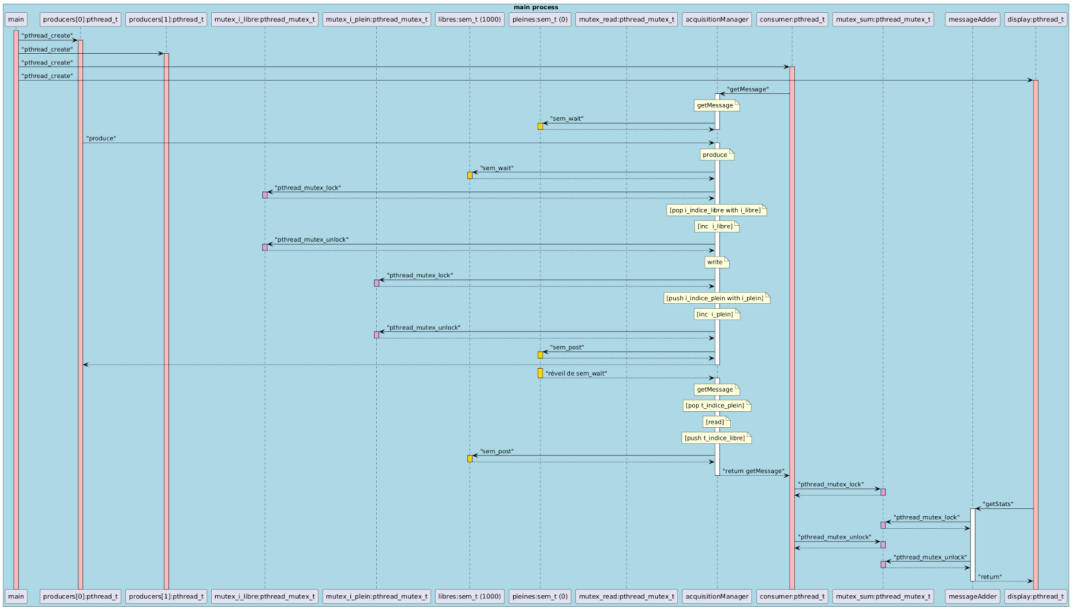
[displayManager]Produced messages: 8, Consumed messages: 8, Messages left: 0
[OK] Checksum validated
[messageDisplay]Message:
[0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 136 144 152 160 168 176 184 192 200 208 216 224 232 240 248 256 264 272 280 288 296 304 312 320 328 336 344 352 360 368 376 384 392 400 408 416 424 432 440 448 456 464 472 480 488 496 504 512 520 528 536 544 552 560 568 576 584 592 600 608 616 624 632 640 648 656 664 672 680 688 696 704 712 720 728 736 744 752 760 768 776 784 792 800 808 816 824 832 840 848 856 864 872 880 888 896 904 912 920 928 936 944 952 960 968 976 984 992 1000 1008 1016 1024 1032 1040 1048 1056 1064 1072 1080 1088 1096 1104 1112 1120 1128 1136 1144 1152 1160 1168 1176 1184 1192 1200 1208 1216 1224 1232 1240 1248 1256 1264 1272 1280 1288 1296 1304 1312 1320 1328 1336 1344 1352 1360 1368 1376 1384 1392 1400 1408 1416 1424 1432 1440 1448 1456 1464 1472 1480 1488 1496 1504 1512 1520 1528 1536 1544 1552 1560 1568 1576 1584 1592 1600 1608 1616 1624 1632 1640 1648 1656 1664 1672 1680 1688 1696 1704 1712 1720 1728 1736 1744 1752 1760 1768 1776 1784 1792 1800 1808 1816 1824 1832 1840 1848 1856 1864 1872 1880 1888 1896 1904 1912 1920 1928 1936 1944 1952 1960 1968 1976 1984 1992 2000 2008 2016 2024 2032 2040]

Nous pouvons observer que le programme fonctionne bien car à la fin des 8 itérations, le tableau de valeur accumulé contient les 256 premiers multiples de 8.

Question 7:

Nous avons rendu cohérent la sortie diagnostic et la sortie en exportant une fonction "*getStats()*" qui utilise le mutex "*mutex_current_sum*" pour protéger également le "*produced_count*" et le "*current_sum*" et renvoie une structure de type stats_t qui contient une copie cohérente de "*produced_count*", "*current_sum*" et surtout de la "*current_sum*". Cette structure est utilisée par le displayManager.

On a également retiré les getProducedCount() et getCurrentSum() de l'interface externe.



Voici le diagramme UML mis à jour avec la cohérence de la sortie.

Question 8:

L'exécution multithread peut engendrer des problèmes lorsqu'il y a des optimisations. volatile interdit les optimisations et évite par exemple qu'une valeur stockée dans un registre par un thread soit ré-utilisée alors qu'elle a été modifiée dans un autre thread.

Question 9:

Les processus POSIX permettent d'avoir des espaces mémoires pleinement séparés (hors des shared memory explicitement demandées). Cette caractéristique permet une meilleure encapsulation des différentes fonctionnalités, et donc un meilleur partitionnement. Par contre, il est plus compliqué de communiquer des données entre les processus comme des sémaphores.

Pour communiquer entre des processus POSIX nous pouvons utiliser des shm qui nécessitent des sémaphores pour la synchronisation ou alors des pipes qui s'occupent eux même du buffering et de la synchronisation. Nous pouvons aussi utiliser d'autres IPC (socket UNIX, files de messages) bien qu'ils soient beaucoup plus lourds.

Question 10:

Les compteurs sont des entiers, qui peuvent être incrémentés ou lus de manière atomique au niveau matériel.
Une api est proposée en C permettant d'exploiter ces fonctionnalités. Elle propose le keyword _Atomic. Si on utilise des entiers atomiques, cela assure que toutes les lectures et modifications soient atomiques, donc pas besoin de mutex.

Voici la sortie du programme utilisant "*acquisitionManagerAtomic.c*", avec pour valeur de mesure l'index de la mesure:

[displayManager] Produced messages: 8, Consumed messages: 8, Messages left: 0
[OK] Checksum validated
[displayManager] Message:
[0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 136 144 152 160 168 176 184 192 200 208 216 224 232 240 248 256 264 272 280 288 296 304 312 320 328 336 344 352 360 368 376 384 392 400 408 416 424 432 440 448 456 464 472 480 488 496 504 512 520 528 536 544 552 560 568 576 584 592 600 608 616 624 632 640 648 656 664 672 680 688 696 704 712 720 728 736 744 752 760 768 776 784 792 800 808 816 824 832 840 848 856 864 872 880 888 896 904 912 920 928 936 944 952 960 968 976 984 992 1000 1008 1016 1024 1032 1040 1048 1056 1064 1072 1080 1088 1096 1104 1112 1120 1128 1136 1144 1152 1160 1168 1176 1184 1192 1200 1208 1216 1224 1232 1240 1248 1256 1264 1272 1280 1288 1296 1304 1312 1320 1328 1336 1344 1352 1360 1368 1376 1384 1392 1400 1408 1416 1424 1432 1440 1448 1456 1464 1472 1480 1488 1496 1504 1512 1520 1528 1536 1544 1552 1560 1568 1576 1584 1592 1600 1608 1616 1624 1632 1640 1648 1656 1664 1672 1680 1688 1696 1704 1712 1720 1728 1736 1744 1752 1760 1768 1776 1784 1792 1800 1808 1816 1824 1832 1840 1848 1856 1864 1872 1880 1888 1896 1904 1912 1920 1928 1936 1944 1952 1960 1968 1976 1984 1992 2000 2008 2016 2024 2032 2040]

Les résultats sont toujours égaux à l'implémentation précédente.

Question 14:

On utilise comme métrique le temps d'exécution du programme qui produit 8 messages. On exécute plusieurs fois le programme et on calcule la médiane de chaque exécution.

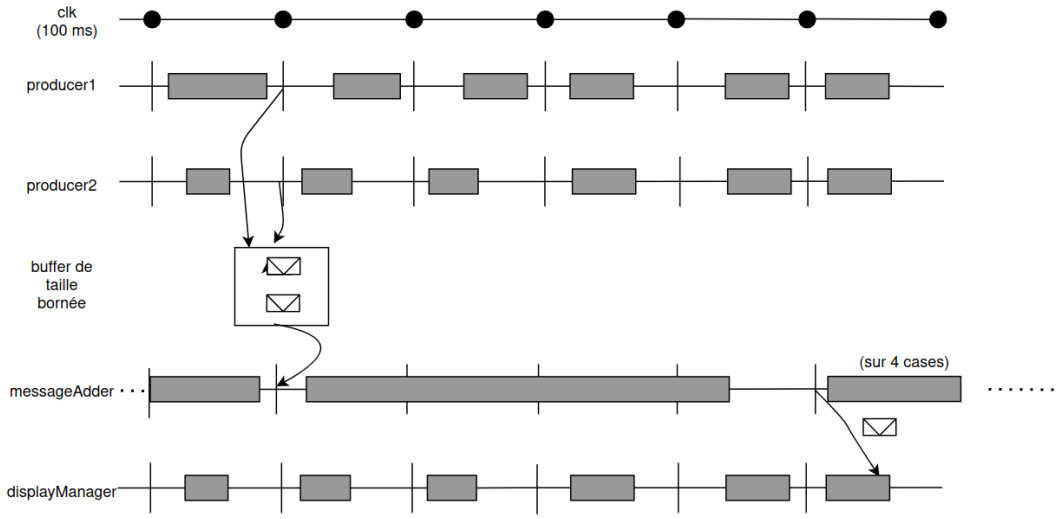
1. Dans la version **POSIX**: **36** microsecondes.
2. Dans la version **Atomic**: **15** microsecondes.
3. Dans la version **TestAndSet**: **38** microsecondes.

La meilleure version est sans conteste Atomic, qui n'a pas d'overhead de synchronisation. Les autres versions utilisent des fonctions système avec nécessairement un overhead. Atomic est basé sur des instructions matérielles bas niveau.

Question 15:

Une approche dirigée par le temps est synchrone. En effet, par exemple, dans psyc, à chaque tick d'horloge on applique toutes les modifications qui ont eu lieu entre les différents ticks. Du point de vue de l'utilisateur, on masque tous les détails d'ordonnancement (qui n'intéressent plus que l'OS) et on masque également tous les détails de ce qui se passe entre les deux ticks.

Question 16:



On a ici bien respecté à la fois l'encapsulation des différents modules, en même temps on respecte les deux exigences. Pour respecter ces deux contraintes, sans fusionner messageAdder et displayManager, il faut décider d'un arbitrage entre les temps d'exécution des additions et du display, qui ensemble doivent faire 5 unités. Ici, on a choisi 4 pour l'addition (la fonction la plus importante) et 1 pour le display.