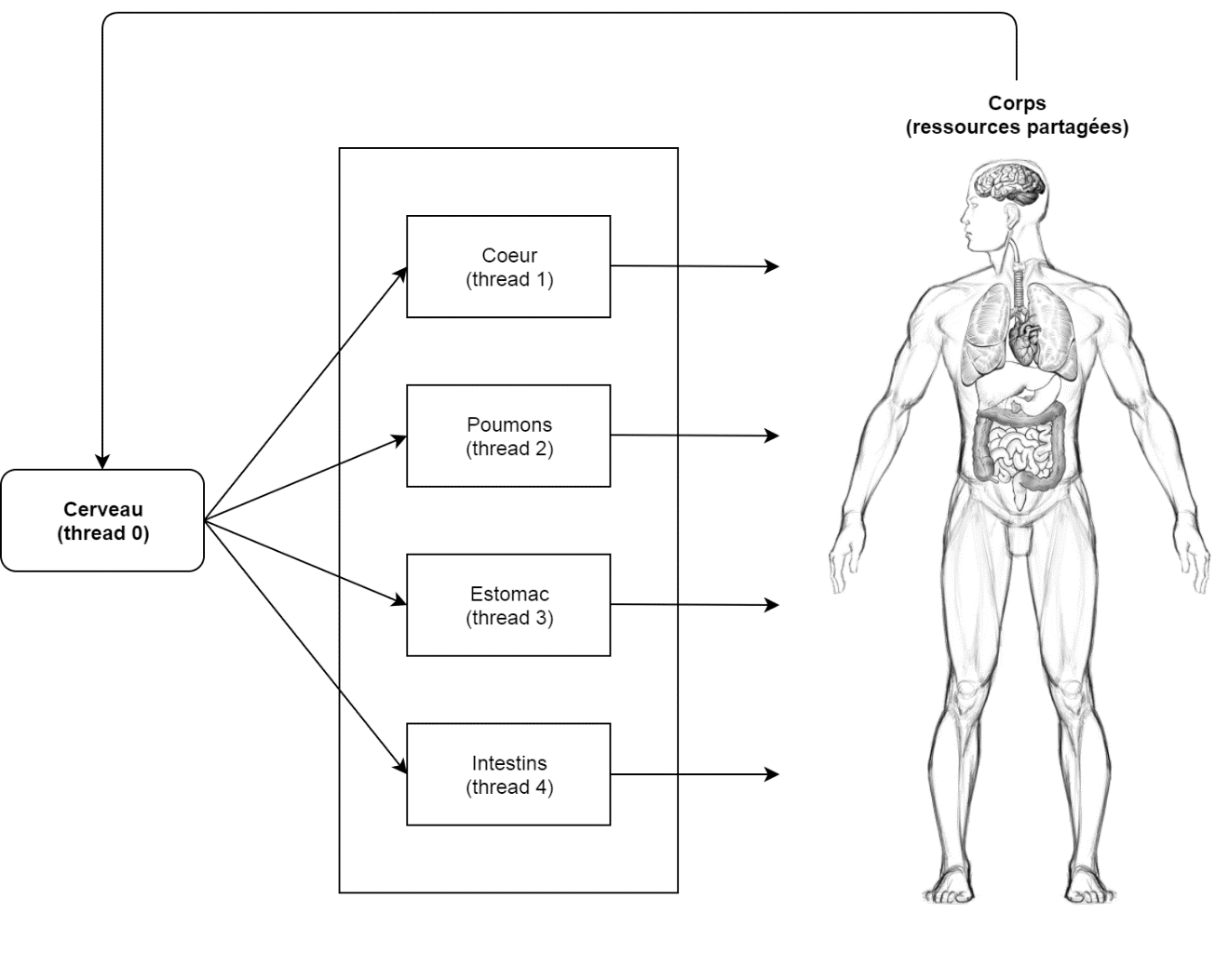
Démarche et recherches

Dans le cadre du cours en parallélisme, nous avons décidé de faire un corps humain minimaliste fonctionnel comme projet final. En effet, nous cherchions une idée originale et de notre cru qui tirerait bien profit de la matière du cours. Ainsi, après y avoir songé pendant plusieurs jours, nous avons réalisé que le cerveau humain envoyait des signaux sous forme d’influx nerveux à différentes parties du corps afin de leur faire effectuer des actions. Les organes, qui sont les principaux acteurs du corps humain, fonctionnent en harmonie mais de manière autonome. Ainsi, chaque organe effectue une tâche bien spécifique qui peut avoir des conséquences ou non sur le fonctionnement des autres organes, mais ce travail s’effectue en même temps et non de manière séquentielle. Autrement dit, chaque organe attend un signal du cerveau pour effectuer sa tâche sans dépendre directement des autres organes. Après avoir fait cette réalisation, nous en sommes venus à la conclusion que c’était un exemple intéressant et quasi-parfait de système parallèle.



En effet, le cerveau lit les données du corps, et envoie aux organes des signaux leur permettant de fonctionner et de réguler les données. Ainsi, chaque organe modifie au sein de leur tâche respective différentes données du corps de manière asynchrone, et le cerveau les lit pour ajuster l’envoi des signaux.

Voici la liste des éléments pris en compte dans le fonctionnement du corps (attributs partagés) ainsi que leur acteur concerné :

* Cerveau : niveau de sommeil
* Cœur : rythme cardiaque
* Poumons : rythme respiratoire et taux d’oxygène dans le sang
* Corps : niveau d’eau et de nutriments

Pour chacune de ces informations, des bornes ont été fixées afin d’assurer la stabilité du système. De plus, une valeur optimale pour chacune d’elles a été déterminée pour permettre la régularisation du système. Voici les valeurs que nous avons choisies (à partir de différentes statistiques trouvées au ligne) pour un sujet âgé d’environ 20 ans :

|  |  |  |  |
| --- | --- | --- | --- |
| Donnée | Min | Optimal | Max |
| Rythme cardiaque | 40 | 100 | 200 |
| Rythme respiratoire | 10 | 15 | 50 |
| Taux d’oxygène | 0.9 | 0.95 | 1 |
| Niveau d’eau | 0.45 | 0.55 | 0.65 |
| Niveau de nutriments | 0.05 | 0.5 | 1 |

Vert : données passives (celles qui sont perturbées et qu’il faut réguler)

Orange : données actives (celles qui permettent de réguler les passives)

À noter que ces valeurs sont approximatives et ne devraient pas être considérées comme étant 100% réalistes. L’essence du travail n’étant pas de nature anatomique, ces valeurs ont peu d’importance dans le cadre du projet.

Cela étant dit, le but principal programme est de réguler les données passives de telle sorte qu’elles s’approchent le plus possible de leur valeur optimale. Ainsi, le cerveau envoie judicieusement des signaux aux organes en fonction des données à réguler, et les organes rétablissent l’ordre progressivement. Toutefois, tout cela ne pourrait arriver sans perturber les données passives. Alors, le programme a été conçu de telle sorte qu’il est possible de les perturber et d’observer les changements dans les données actives. D’un point de vue graphique, les données actives sont directement liées aux organes et peuvent être visualisées.

Signaux

Tel que mentionné précédemment, chaque organe s’exécute sur un fil d’exécution différent (à l’exception du cerveau qui est un cas à part, voir plus loin). Ainsi, chacun d’entre eux attend un signal de la part du cerveau avant d’exécuter leur tâche bien précise et se remettent en attente ensuite (jusqu’à la fermeture du programme). Pour faire parvenir les signaux entre le cerveau et les autres organes, la classe ThreadSignaler a été implémentée. Ainsi, chaque organe à l’excepté du cerveau possède son propre fil d’exécution et son propre ThreadSignaler, permettant ainsi l’envoi de signaux et l’exécution des tâches. Cette classe fait usage des variables de condition en C++ (std::condition\_variable), qui permettent de bloquer un fil d’exécution jusqu’à ce qu’un signal soit reçu. Ainsi, la classe ThreadSignaler est implémentée comme suit :

(pseudocode)

Comme on peut le constater, trois éléments ont été utilisés :

1. Un mutex : permet de s’assurer que les méthodes ne peuvent être appelées que par un fil d’exécution à la fois (empêche l’appel de notify() et wait() en même temps)
2. Une variable de condition : permet de notifier un fil d’exécution en attente sur la même variable de condition
3. Une variable booléenne : permet d’éviter les faux réveils (voir plus loin)

Il est important de mentionner qu’en C++, les variables de conditions sont implémentées de telle sorte qu’un appel à notify\_one() lorsqu’aucun fil d’exécution n’attend est équivalent à ne rien faire. Pour comprendre cette notion, voici une comparaison de trois mécanismes similaires en C# :

1. Monitor : lorsqu’un Monitor.Pulse() est émis, seuls les fils d’exécution en attente lors de l’appel sont affectés. Analogie : si quelqu’un crie et que personne ne l’entend, la personne devra recrier pour se faire entendre (si les personnes sont à l’écoute).
2. AutoResetEvent : lorsque Set() est appelé, le signal est levé jusqu’à ce qu’un fil d’exécution se mette en attente. Analogie : un drapeau est planté jusqu’à ce que quelqu’un vienne le chercher.
3. ManualResetEvent : lorsque Set() est appelé, le signal est levé jusqu’à ce que Reset() soit appelé, laissant ainsi passer tous les fils d’exécution de mettant en attente durant cette période. Analogie : porte que l’on ouvre ou ferme pour laisser passer des clients.

Dans notre cas, les variables de condition en C++ sont à peu près équivalentes aux moniteurs en C# (d’un point de vue émission et réception de signaux). En effet, une implémentation similaire de ThreadSignaler en C# pourrait ressembler à ceci :

<https://stackoverflow.com/questions/15657637/condition-variables-c-net>

Quant aux faux réveils, c’est un phénomène qui se produit lorsqu’un fil d’exécution en attente reçoit un signal ne provenant pas du programme lui-même, le réveillant ainsi de manière fautive. Cela se produit parfois avec les fils d’exécution POSIX et Windows afin d’accélérer le traitement des variables de condition. Même si ce phénomène est rare, il n’est généralement pas souhaitable et peut être éviter en utilisant une variable booléenne qui indique si le signal provient vraiment du programme lui-même ou non. Ainsi, lors de l’appel à notify(), le flag est levé, et lorsque la variable de condition est réveillée, on vérifie se le flag est bel et bien levé, sinon on relance une attente.

Ressources partagées

Quant aux ressources partagées, ces dernières doivent être synchroniser afin d’assurer un fonctionnement optimal et fiable du corps. Pour ce faire, les données du corps ont été déclarées comme étant atomiques dans une structure nommée BodyInfo. En effet, la librairie standard C++ fournit une interface qui permet d’utiliser et de manipuler atomiquement certains types de données (std::atomic<T>). Malheureusement, les seuls types pouvant être manipulés nativement de manière atomique sont les types intégraux (char, int, etc.). Dans le cas d’une classe T quelconque, seule l’affectation est atomique. Dans le cas des nombres en virgule flottante (float, double), les opérations atomiques sont supportées par le langage qu’à partir de C++20. Heureusement, il est possible d’implémenter ces opérations nous-mêmes pour des versions antérieures. C’est pourquoi nous avons créé une classe nommée AtomicNumber permettant de le faire. La classe std::atomic<T> fournit deux méthodes permettant d’échanger la valeur de la variable pour une autre valeur. Ces méthodes s’appellent compare\_exchange\_weak et compare\_exchange\_strong. Tout comme les variables de condition, ces échanges sont soumis à un problème similaire aux faux réveils où la valeur attendue n’est pas exactement la bonne même si la méthode indique le contraire (rare mais possible). Ainsi, il faut boucler pour s’assurer que la valeur est bel et bien la bonne avec compare\_exchange\_weak, ou tout simplement utiliser compare\_exchange\_strong qui fait exactement le même travail (plus lent). Dans notre cas, l’opération est effectuée jusqu’à ce que la valeur retournée soit la bonne, rendant ainsi l’opération atomique.

Mode de déploiement

Références