Exercices d'introduction aux calculs parallèles

Partie 2

Parallélisme et concurrence avec Python

Contrôleur de température (& Pression)
Déplacements d'un Robot
Restaurant

Game of Life

CPE - Mai-Juin 2025

(Version élèves)

ASG

I Projets CPE : Python Concurrent

- Ce document est la suite (partie 2) des exercices "Projets". Il contient plusieurs sujets :
  - Contrôleur de température / Pression (**, 5 pts)
  - Simulation des déplacements d'un Robot (***/****, 6 pts)
  - Simulation des serveurs/client/cuisiniers d'un restaurant (***/****, 6 pts)
  - Game of life

II Réalisation d'une système muti-tâche de contrôle de Température et Pression

Difficulté : ** **(5 points)**

On considère le système (*temps réel embarqué*) simple suivant :

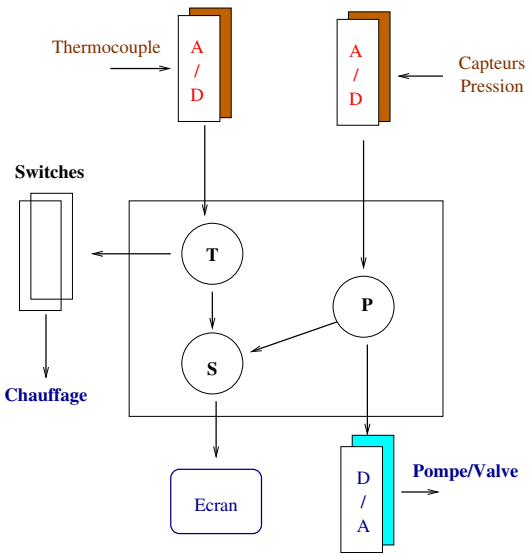
• Un **processus T** (température) lit les valeurs d'un ensemble de thermocouples (capteurs de température) par l'intermédiaire d'un convertisseur analogique-numérique, ADC.

☞ Noter que dans cette application, les valeurs (température, pression, ...) seront numériques et donc aucun convertisseur "analogique-digital" ne sera représenté. Ils sont simplement cités ici dans un but de réalisme du dispositif.

➙ T envoie par commande les changements appropriés (allumer, éteindre, baisser, monter, ...) à un chauffage par l'intermédiaire d'un commutateur à commande numérique.

• Le **processus P** (pression) a une fonction similaire pour la pression (il emploie un convertisseur numérique-analogique, DAC).

• T et P doivent communiquer des données au **processus S** (screen, affichage), qui affiche différentes mesures à destination d'un utilisateur / opérateur par l'intermédiaire d'un écran d'affichage (le boîtier que l'on retrouve sur le mur dans un lieu climatisé).



• Notez que P, T et S sont les entités actives (processus).

• S reçoit des valeurs de la part de T et P pour affichage. Il affiche également les *consignes* (température / pression min et max).

• L'écran d'affichage est une ressource unique et protégée. Seul le processus S a le droit d'écrire à l'écran.

• Les consignes sont des paramètres fixés dès le début de l'application.

• L'objectif global de ce système temps réel embarqué est de maintenir la température et la pression d'un certain processus chimique dans des limites définies.

• Un vrai système de ce type serait clairement plus complexe, permettant par exemple à l'opérateur de modifier les limites (consignes). Cette partie est laissée en **bonus**.

• Le but de ce système est de conserver la température et la pression d'un processus chimique dans des limites spécifiées.

• Habituellement (dans l'industrie), deux approches sont possibles pour réaliser une telle application.

1- Une approche synchrone (et séquentielle) qui ne crée pas de processus. Dans une itération et de manière séquentielle, le contrôleur interroge un par un les capteurs, calcule les ajustements (par rapport aux consignes), envoie des commandes et procède aux affichages ;

2- Approche asynchrone avec des entités parallèles (comme dans la réalité où les éléments du dispositifs fonctionnent en parallèle). Nous réalisons cette seconde approche ici.

• On distingue plusieurs entités concurrentes :

- Gestionnaire de la température (T)
- Gestionnaire de la pression(P)
- Gestionnaire du Chauffage
- Gestionnaire de la Pompe
- La tâche Ecran (S)
- Un **contrôleur** (le cerveau !) pour coordonnée l'ensemble

• Ci-dessous, quelques éléments (en pseudo code) sont présentés pour vous aider à avancer. Vous n'êtes pas obligé de les suivre de manière stricte.
Par exemple, la tâche **Contrôleur** traite le chauffage en même temps que la pression. Vous pouvez séparer les deux !
De même, la tache **Température** ci-dessous est en fait un contrôleur de la partie température du dispositif.
Dans le but de simplifier, vous pouvez la remplacer par une tâche **Capteur_Temp** (de température) qui modifiera la valeur Mem_T (température).
Pour les écritures, donner l'exclusivité des affichages à la tâche **Ecran** (S : Screen).
Dans ce cas, les autres tâches enverront leurs messages (à afficher) via une queue (ou une mémoire partagée) à la tache **Ecran**. Cette manière de donner (à la tache Ecran) l'accès exclusif à l'écran convient mieux aux interfaces graphiques (par exemple, TkInter, Qt, etc...) où le partage du canevas de dessin est délicat.

N.B. : ci-dessus, "– commentaires" désigne le début d'un commentaire (comme en langage ADA).

Déclarations :

Ver : Verrou – cf. TAS

Seuil_T, Seuil_P : réel ← ..

go_pompe : bool ← faux

go_chauffage : bool ← faux

mem_xx : mémoire partagée

☞ mémoire partagée :

si *thread* (ou task ADA/C/Java) utilisés alors une variable globale si non un *shmem*.

➙ Certains langages proposent des variables **protégées** = variable globale + Verrou mutex

Tâche Contrôleur :

Répéter toutes les X secondes

verrouiller(Ver) ;

$T \leftarrow Mem_T$ $P \leftarrow Mem_P$

libérer(Ver) ;

Si ($T > Seuil\_T$)

go_chauffage ← faux – pour le chauffage

Si ($P > Seuil\_P$)

go_pompe ← vrai – pour la Pompe

Sinon go_pompe ← faux

Sinon Si ($T < Seuil\_T$)

go_pompe ← vrai

go_chauffage ← vrai

Sinon – $T = Seuil\_T$

go_chauffage ← faux

Si ($P > Seuil\_P$) go_pompe ← vrai

Sinon go_pompe ← faux

Fin Répéter

Tâche Chauffage :

Répéter toutes les Z secondes

Si (go_chauffage) Alors

"mettre en route"

Sinon "arrêter"

Fin si

Fin Répéter

• En général, le contrôleur crée les tâches après sa propre création.

Tâche Température :

Répéter toutes les S secondes

Réel : lire la valeur V sur le capteur et convertir_AD(V,T)

Simulation : générer aléatoirement une valeur T pour le capteur

verrouiller(Ver) ;

Ecrire(T,Mem_T)

libérer(Ver) ;

Fin Répéter

Tâche Pression :

Répéter toutes les U secondes

Réel : lire la valeur V sur le capteur et convertir_AD(V,P)

Simulation : générer aléatoirement une valeur P pour le capteur

verrouiller(Ver) ;

Ecrire(P,Mem_P)

libérer(Ver) ;

Fin Répéter

Tâche Pompe :

Répéter toutes les Z secondes

Si (go_pompe) Alors

"mettre en route"

Sinon "arrêter"

Fin si

Fin Répéter

Tâche Ecran :

Répéter

verrouiller(Ver) ;

$T \leftarrow Mem_T$

$P \leftarrow Mem_P$

libérer(Ver) ;

écrire T et P

Fin Répéter

• La gestion par les booléennes *go_pompe*, *go_chauffage* peut être remplacée par le mécanisme d'évènement (*Attendre, Signaler*) :

➙ la tâche Pompe fera *Attendre(go_pompe)* conjugué avec *Signaler(go_pompe)* effectué par le Contrôleur.

• Ces booléennes n'ont pas besoin d'un accès en *mutex* car le *contrôleur* y écrit et Pompe (ou Chauffage) lisent.

III Simulation des déplacements d'un Robot

Difficulté : ***/***** , **(6 points)**

☞ La partie graphique doit impérativement être réalisée sous **TkInter**. Les versions qui existent sur le WEB ne sont pas acceptées (ne sont pas réalisées avec Tkinter) !

- Un robot avec les caractéristiques suivants
  - — Pas de but particulier : avancer et éviter les obstacles
  - — Plusieurs capteurs : infra rouge (IR) sur les 2 côtés, sonar (US) frontal, de contact (Bumper) frontal
  - — Les actions sur les servo moteurs : *avancer*, *reculer*, *tourner à gauche/droite*
  - — Le comportement par défaut est : *avancer*
  - — Un écran d'affichage de l'état
- Principes : lecture des capteurs

Déclarations :

Ver : Verrou – cf. TAS

les Distances : réel ← ..

les Drepeaux : bool ← faux

mem_xx : mémoire partagée

☞ mémoire partagée :

si *thread* (ou task ADA, thread Java/C) utilisés alors une variable globale si non un *shmem*.

➙ Certains langages proposent des variables **protégées** = variable globale + Verrou mutex

Tâche Controleur :

Répéter toutes les X secondes

Commande ← "avancer"

Drapeau ← faux

Si (Drapeau_IR) Alors

  Commande ← Cmd_IR

  Drapeau ← Drapeau_IR

Si (Drapeau_US) Alors

  Commande ← Cmd_US

  Drapeau ← Drapeau_US

Si (Drapeau_BU) Alors

  Commande ← Cmd_BU

  Drapeau ← Drapeau_BU

Transmettre $Commande$ aux servos

verrouiller(Ver) ;

$mem\_Cmd \leftarrow Commande$

$mem\_Flag \leftarrow Drapeau$

libérer(Ver) ;

Fin Répéter

Tâche Ecran :

Répéter tou sles A secondes

verrouiller(Ver) ;

$C \leftarrow mem\_Cmd$

$F \leftarrow mem\_Flag$

libérer(Ver) ;

écrire C et F

Fin Répéter

- En général, le contrôleur crée les tâches après sa propre création.

Tâche IR :

Répéter toutes les S secondes

Réel : lire la valeur Vg sur le capteur gauche et convertir_AD(Vg,Dg)

Simulation : générer aléatoirement une valeur Dg pour le capteur

De même pour Vd / Dd

Si $Dg < d$ OU $Dd < d$ Alors

Drapeau_IR ← vrai

Si $Dg < d$ ET $Dd < d$ Alors

Cmd_IR ← "reculer"

Sinon Si $Dg < d$ Alors

Cmd_IR ← "à gauche"

Sinon Cmd_IR ← "à droite"

Sinon Drapeau_IR ← faux

Fin Répéter

Tâche US :

Répéter toutes les K secondes

Réel : lire la valeur V sur le capteur gauche et convertir_AD(V,D)

Simulation : générer aléatoirement une valeur D pour le capteur

Si $D < d$ Alors

Drapeau_US ← vrai

Cmd_US ← "reculer"

Sinon Drapeau_US ← faux

Fin Répéter

Tâche Bumper :

Répéter toutes les Z secondes (Z petit)

Si (contact=1) Alors

Drapeau_BU ← vrai

Cmd_BU ← "reculer"

Sinon Drapeau_BU ← faux

Fin Répéter

Remarque sur "Répéter toutes les X milli/micro/nanosecondes" :
Un moyen simple d'implanter ce délai :

Next ← temps actuel (clock)

Répéter

Actions

$Next \leftarrow Next + X$

delay until next

Fin Répéter

- Si *delay* non disponible :

Temps ← temps actuel (clock)

Répéter

Actions

Next ← temps actuel (clock)

Reste ← X - (Next - Temps)

Attendre(Reste) – e.g. usleep/sleep

Temps ← Next

Fin Répéter

➞ Bien entendu, $Reste > 0$ sinon, le système n'est pas RT!!

IV Un système muti-tâches de simulation d'un restaurant

Difficulté : ***/***** , **(6 points)**

On considère le système (*temps réel*) simple suivant qui :

1. simule des commandes de clients dans un restaurant
2. un certains nombre de serveurs en salle enregistrent ces commandes et les transmettent à la cuisine pour préparation
3. après leur préparation, les serveurs délivrent ces commandes aux clients

Dans la version de base, on n'identifie pas de cuisinier et ce sont les serveurs qui simulent la préparation des commandes (voir plus bas pour la version étendue).
Prévoir :

- s processus *serveur*. P. Ex. $s = 5$
- un processus *clients* qui simulera aléatoirement les commandes des clients selon une loi uniforme. Ce processus émettra une commande aléatoire toutes les p. ex. $3..10$ $secondes$ à l'adresse des serveurs.
- un processus *major_dHomme* qui s'occupera des affichages à l'écran
- un tampon de taille (p. ex.) 50 contiendra les commandes des clients ; les serveurs prélèvent des commandes de ce tableau
- une commande d'un client sera constituée d'un identifiant client (un entier) et une lettre $A..Z$ qui représentera le menu commandé

En l'absence d'interface graphique, on utilisera le module **curses** de Python que l'on a déjà utilisé dans l'exemple cours de chevaux. On affichera ainsi à l'écran les informations suivants :

- les commandes des clients (les paires *(id, menu)*) dès leur émission
- le serveur qui prend cette commande en charge et simule sa préparation (par un délai)
- le client qui reçoit sa commande préparée

☞ Les informations sont affichées exclusivement par le processus *major_dHomme*.
Un exemple d'affichage à l'écran :

Le serveur 1 **traite la commande** (id_i, C_i) (ou rien si pas de commande traité par ce serveur)

....

Le serveur s **traite la commande** (id_j, C_j)

Les commandes clients en attente : [(id_i, C_i) **,** (id_j, C_j) **...**(id_k, C_k) **]**

Nombres de commandes attente : 5

Commande (id_u, U) **est servie au client**

Aller plus loin (Bonus) :

Ajouter un certains nombre de cuisiniers (en cuisine) qui préparent ces commandes et avertissent les serveurs. Le serveur qui avait enregistré la commande la délivre au client qui a commandée.

Ajouter à l aversion de base :

- c processus *cuisto*. P. Ex. $c = 2$
- Modifier les affichage et présenter le cuisinier qui traite la commande.

  Le contenu de l'écran sera augmenté des lignes :

  Le cuisiner 1 **prépare la commande** $(id_1, A, serveur_1)$ (ou rien si pas de commande traité par ce cuisinier)

  ....

  Le cuisiner c **prépare la commande** $(id_p, P, serveur_p)$

V Game of Life

Difficulté : *** , **(5 points)**

Réaliser le jeu suivant dans une version **concurrente** avec les mécanismes de base graphique (*screen* comme dans la course Hippique).

Il s'agit d'une grille (matrice de taille d'au moins $15x15$) dont les cases représentent soit un "être" vivant soit rien. L'état d'une case peut être modifié en fonction de son voisinage selon les règles décrites ci-dessous.

Extrait de l'énoncé d'origine :

• The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead.

• Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur :

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

• The initial pattern constitutes the seed of the system.

• The first generation is created by applying the above rules simultaneously to every cell in the seed-births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one).

• The rules continue to be applied repeatedly to create further generations.

ASG, Mai 2025