

# Programmation concurrente

## Jackpot

David Gonzalez - Claudio Sousa

1 février 2017

# 1 Introduction

Ce TP de deuxième année consiste à implémenter une machine à sous de casino multi-tâches.

## 1.1 Spécification fonctionnelle

Chaque partie débute avec l'insertion d'une pièce. Ensuite, 3 roues tournent à des vitesses différentes. Chaque roue est arrêtée consécutivement (de gauche à droite) soit manuellement par l'utilisateur, soit automatiquement après 3 secondes.

Lorsque toutes les roues sont arrêtées, le résultat est calculé et soustrait à la caisse de la machine. Les résultats possibles sont :

- aucun chiffre identique (perdu) ;
- 2 chiffres identiques (2 pièces gagnées) ;
- 3 chiffres identiques ou plus (moitié des pièces en caisse).

Ce résultat est affiché pendant 5 secondes, puis on attend à nouveau l'insertion d'une pièce.

### 1.1.1 Entrées

Les entrées sont gérées à l'aide de signaux générés par la console avec les touches suivantes :

- CTRL-Z (SIGTSTP) : insertion d'une pièce ;
- CTRL-C (SIGINT) : arrêt d'une roue manuellement ;
- CTRL-\ (SIGQUIT) : arrêt du jeu.

### 1.1.2 Threads

Les threads sont divisés ainsi :

- 1 thread pour le contrôleur du jeu (qui sera le seul à recevoir les signaux) ;
- 1 thread pour l'affichage ;
- N threads, 1 pour chaque roue.

## 2 Development

### 2.1 Architecture

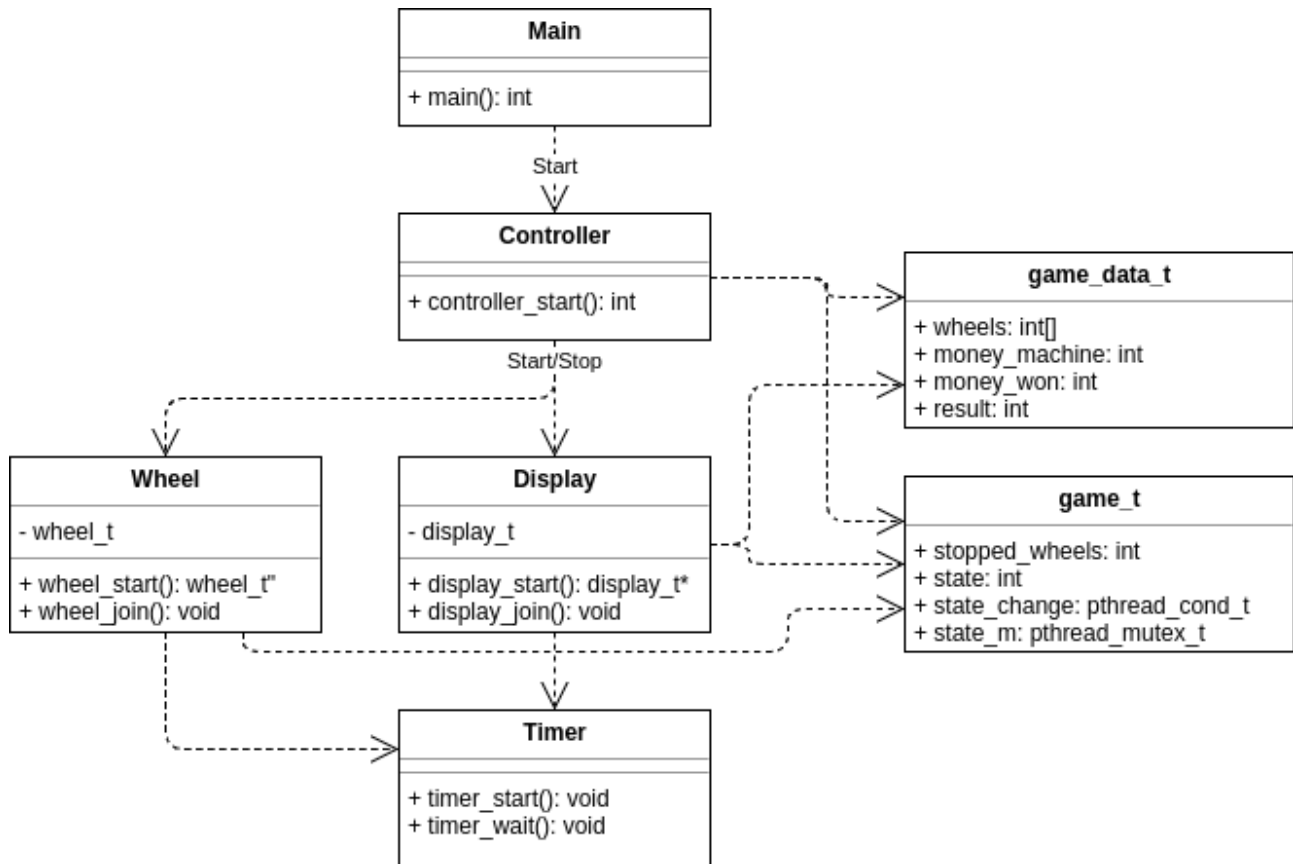


FIGURE 1 – Architecture du Jackpot

#### 2.1.1 `game_t` et `game_data_t`

Ces deux structures représentent toutes les données partagées entre les 3 modules principaux, qui sont : *Controller*, *Wheel*, *Display*.

La première (*game\_t*) contient les données d'état des roues ainsi que les données de synchronisation.

La deuxième (*game\_data\_t*) contient les données du jeu et de la partie : la valeur de chaque roue, l'argent en caisse et l'état final du jeu (gagné, perdu).

#### 2.1.2 `Main`

La fonction principale du programme ne fait que lancer le *Controller*.

### 2.1.3 Controller

Ce module est le coeur du programme. Il est responsable d'instancier le module *Display* ainsi que toutes les instances de *Wheel* (3 par défaut). Il est également chargé de contrôler l'avancement du jeu selon la machine d'état ci-dessous :

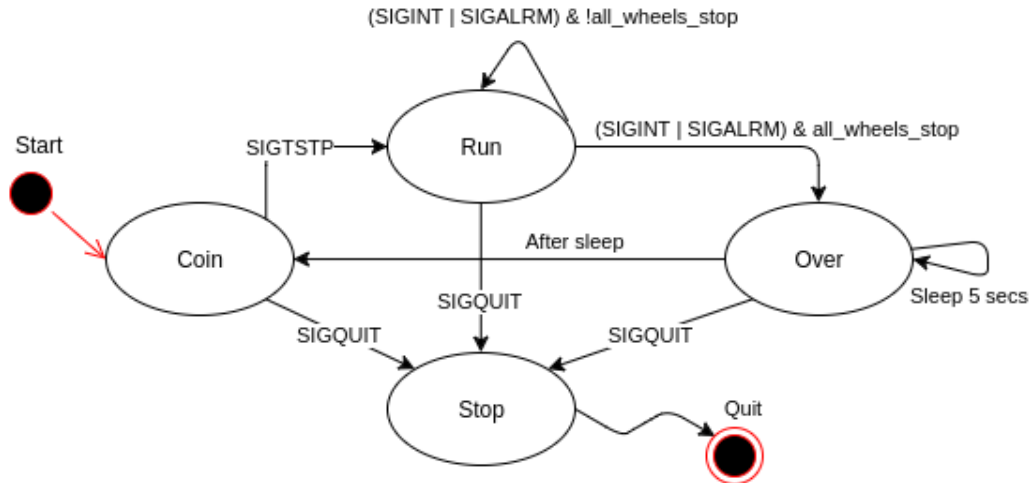


FIGURE 2 – Machine d'état du contrôleur

- *COIN* : état lorsque le jeu est en attente d'une pièce ;
- *RUN* : état lorsque les roues tournent ;
- *OVER* : état lorsque la partie est terminée ;
- *STOP* : état lorsqu'on quitte le jeu.

Comme le dit la spécification, les entrées utilisateurs sont gérées par des signaux. Le *Controller* est donc le seul module (et *thread*) à recevoir et à traiter ces signaux.

Les communications sont faites au travers des deux structures citées (*game\_t* et *game\_data\_t*), qu'il est chargé d'instancier et d'initialiser.

Au niveau des *threads*, le *Controller* crée lui-même son propre *thread* durant son initialisation.

### 2.1.4 Wheel

Le module *Wheel* est responsable de faire tourner 1 roue (changer le chiffre) dans un *thread* séparé à une certaine fréquence.

### 2.1.5 Display

Le module *Display* a le rôle de gérer l'affichage du jeu dans son *thread*. Donc :

- le message de début de partie ;
- les roues lorsque la partie est en cours ;
- le résultat de cette partie ;
- le message lorsqu'on quitte le jeu.

### 2.1.6 Timer

Ce module contient deux fonctions utilitaires permettant de mesurer le temps et d'attendre à une certaine fréquence.

## 2.2 Flux d'exécution et synchronisation

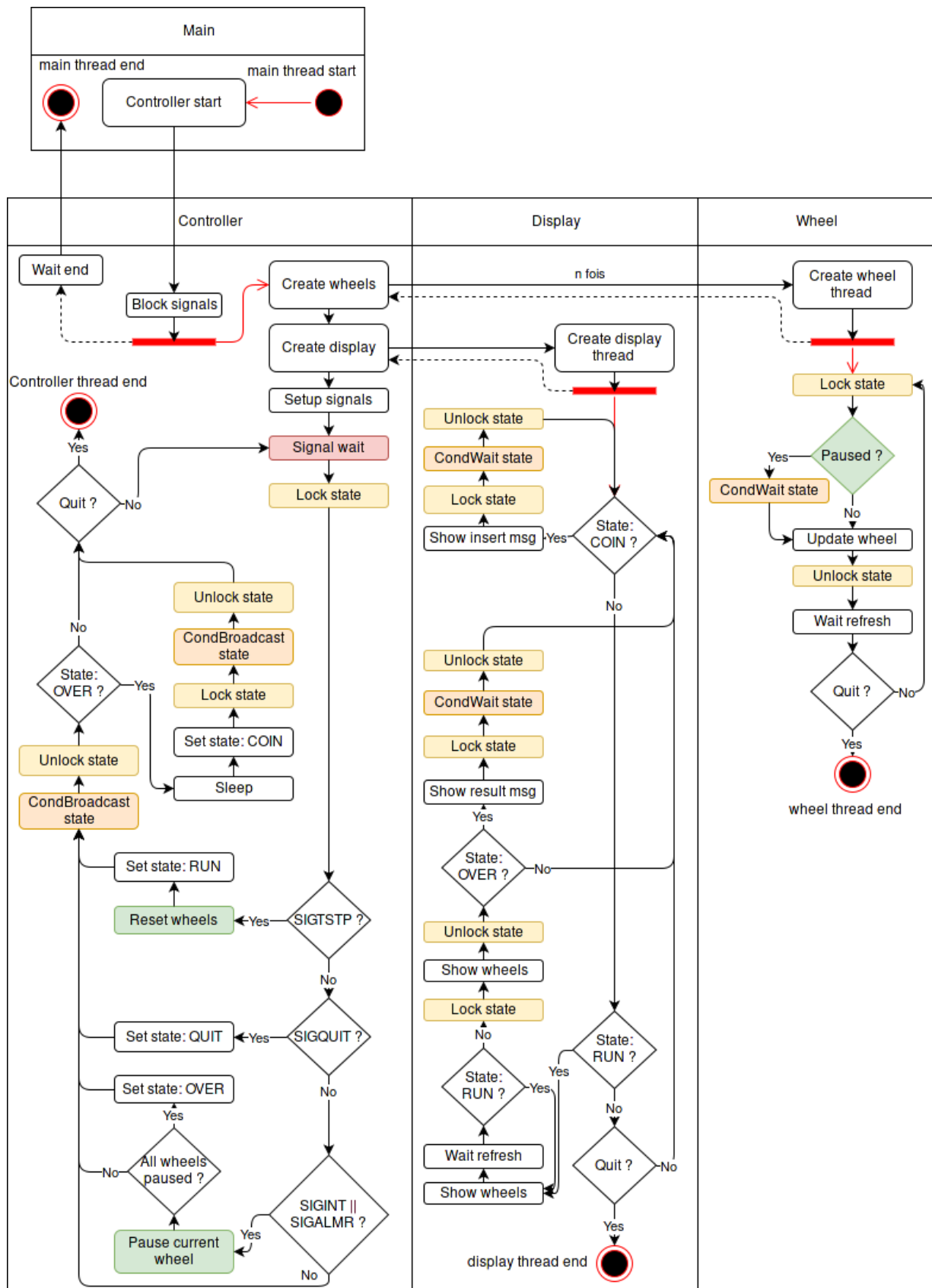


FIGURE 3 – Flux d'exécution et synchronisation

Le diagramme ci-dessus décrit le flux d'exécution du programme par les différents *threads*.

### 2.2.1 Main

Le *thread* principal commence par lancer le module *Controller*. Avant de créer son *thread*, le *Controller* bloque tous les signaux afin que tous les autres *threads* créés après héritent de cette configuration.

S'ensuit la création du *thread* pour le *Controller*.

Le *thread* principal finit sa course par l'attente de la fin du *thread* créé précédemment.

### 2.2.2 Controller

Le *Controller* commence par initialiser les structures de données et lance les modules *Wheel* ainsi que *Display*. La période d'initialisation se termine par la configuration des signaux à traiter.

Ensuite, il rentre dans la boucle du traitement des signaux. C'est dans celle-ci que la machine d'état (voir figure 2) est configurée. En effet, 4 signaux différents peuvent être reçus :

- SIGTSTP : reinitialise l'état des roues, insert une pièce et passe à l'état *RUN* ;
- SIGQUIT : configure le bit de sortie du programme afin que tous les *threads* quittent ;
- SIGINT ou SIGALRM : met en pause la roue courante, si toutes les roue sont en pause, alors l'état *OVER* est mis.

Tous les signaux impliquant un changement d'état, tous les *threads* qui attendent un changement d'état sont réveillés.

Finalement, si l'état est *OVER* en fin de boucle, alors le *Controller* calcule le résultat du jeu, attends les 5 secondes pour l'affichage de celui-ci, passe à l'état *COIN* et réveille à nouveau les *threads*.

La boucle continue indéfiniment tant que le bit de sorti n'est pas configuré.

### 2.2.3 Wheel

Lors du lancement d'une roue, le *thread* du *Controller* commence par créer le *thread* de cette roue en lui passant en paramètre la structure de synchronisation ainsi que le chiffre que la roue va traiter. Le *thread* nouvellement créé rentre immédiatement dans sa boucle qui s'exécute comme suit :

Il commence par vérifier qu'il n'est pas en pause. Si oui, alors il se met en attente jusqu'au prochain changement d'état. Sinon, il continue sa route en mettant à jour la valeur de la roue puis attends sa fréquence de rafraichissement.

Le flux ci-dessus s'exécute indéfiniment tant que la bit de sorti n'est pas configuré.

### 2.2.4 Display

Lorsque le *thread* du *Controller* initialise le module *Display*, un *thread* est créé pour lui. Sa boucle d'exécution est basée sur la machine d'état :

- *COIN* : simplement affiche le message et attends le changement d'état ;
- *RUN* : tant que cet état est actif, affiche les roues et attends sa fréquence de rafraichissement.  
Lorsque le jeu passe en état *OVER*, affiche le résultat et attends l'état suivant.

Comme pour les autres *threads*, la boucle ne se termine pas tant que la bit de sorti n'est pas configuré.

## **2.3 Méthodologie de travail**

### **2.3.1 Répartition du travail**

Ce travail a été effectué à deux.

Une réflexion initiale commune a permis de concevoir les fondations du programme :

- architecture (modules et interface publique) ;
- flux d'exécution (machine d'état et synchronisation).

L'implémentation de cette réflexion a été divisée en parties égales, chacune implémentée par l'un et révisée par l'autre.