HARIRI Safae , IDBELOUCH Salah

Projet PPC: Cambiecolo

Cambiecolo is a card game with the goal of presenting a hand of 5 cards of the same transport means , we have five types of transport :airplane, car, train, bike and shoes. The player who succeeds is awarded the points of the transport they put together.

To implement a multi-process model of the Cambiecolo game and to better visualize the concurrency and the parallelism that we saw in the PPC course .
The game will be in the form of one player interacting with the machine and the other players are going to be bots and in our case “processes”.
We had thought of more players interacting with the machine round by round , but in this case we’re not going to see the parallelism nor the concurrency that we want to implement so we we proceeded as it follows:

The game starts by launching the first process “game” that initiates the game session by asking the player (in the console) how many players are going to play(between 3 and 5) .
By the input given , the game process will create as many child processes as players we have. I’m

As soon as the game starts , cards are distributed to the main player (player interacting with the machine) and the bots (processes) by the function deal in class deck(shown in the second page) .Each player’s going to check the hand of cards that he has and start exchanging by number of cards they offer , from 1 to 3 identical cards without showing them.
In our code , the offers are going to be stored in a global array as a shared memory called offers where all the players can check for offers .Our array is going to contain the number of cards to offer or to take(integers from 1 to 3).

We’re going to create message queues to communicate between processes, with the type of message being an integer from 1 to 3 referring to the number of cards to exchange and the message inside is going to be a string referring to the type of the card (train, bike or airplane…).
So ,for example if the first player wants to exchange one “train” card, he’s going to look for the offers in our shared memory exactly for 1 card offer , if he found it he’s going to call message.queue.receive(type 1) , and if there’ s no offers , he’s going to call a message.queue.send(message= type of card , type 1) to make one.
It’ s going to be the same implementation if the player wants to exchange two or three cards at a time.

To synchronize the processes and to manage the access to the offers,we‘re going to use a mutex for the offers , whenever a player wants to make or take an offer, he ‘s going to lock the shared memory Lock_offre.acquire() and after finishing the action he releases the lock so other players can get in.

For the main player ,the process main player is going to create a thread where it can interact with the real player to retrieve the inputs and to print from and to the console ,where he can see and choose the offers displayed in the console .
Or for the bots , we're going to make a function make_offer , that makes an offer based on the hand of cards of the bot , basically it 's going to choose the card that has the minimum occurrence in the hand and make it as an offer.

For the bell ,it's also a shared memory, we're going to implement it as a global boolean variable with the value true .So all the game code is going to be under the condition while Bell , and once a player have 5 cards of the same suit , he's going to ring the bell by changing the value of bell to False to end the game (get out of the loop). The access to the bell is going to be protected by a second lock Lock_bell. After the bell is rung , we return the winner , and discard all players' processes by sending a SIGKILL signal from the game process .

Pseudo-code :

For the **shared memory** , we're going to implement it a it follows:

- offers = array.array('i', range(Players Nb))

For the **locks** ,it' s going to be a lock for each player trying to get into the shared memory.

- Mutex_player = [Mutex() for i in range(Nb of players)] and another mutex for the bell: Mutex_Bell

For the **message queues** , we 're going to implement it as it follows:

- **import** sysv_ipc
- key **=** 128
- mq **=** sysv_ipc.MessageQueue(key, sysv_ipc.IPC_CREAT)
- **mq.receive**([block = True, [type = nbre of cards(1,2,3]]) to receive the offer.
- **mq.send**(message(type of card), [block = True, [type = (1,2,3]])to send the offer

For the **processes** , we're going to implement them as follows:

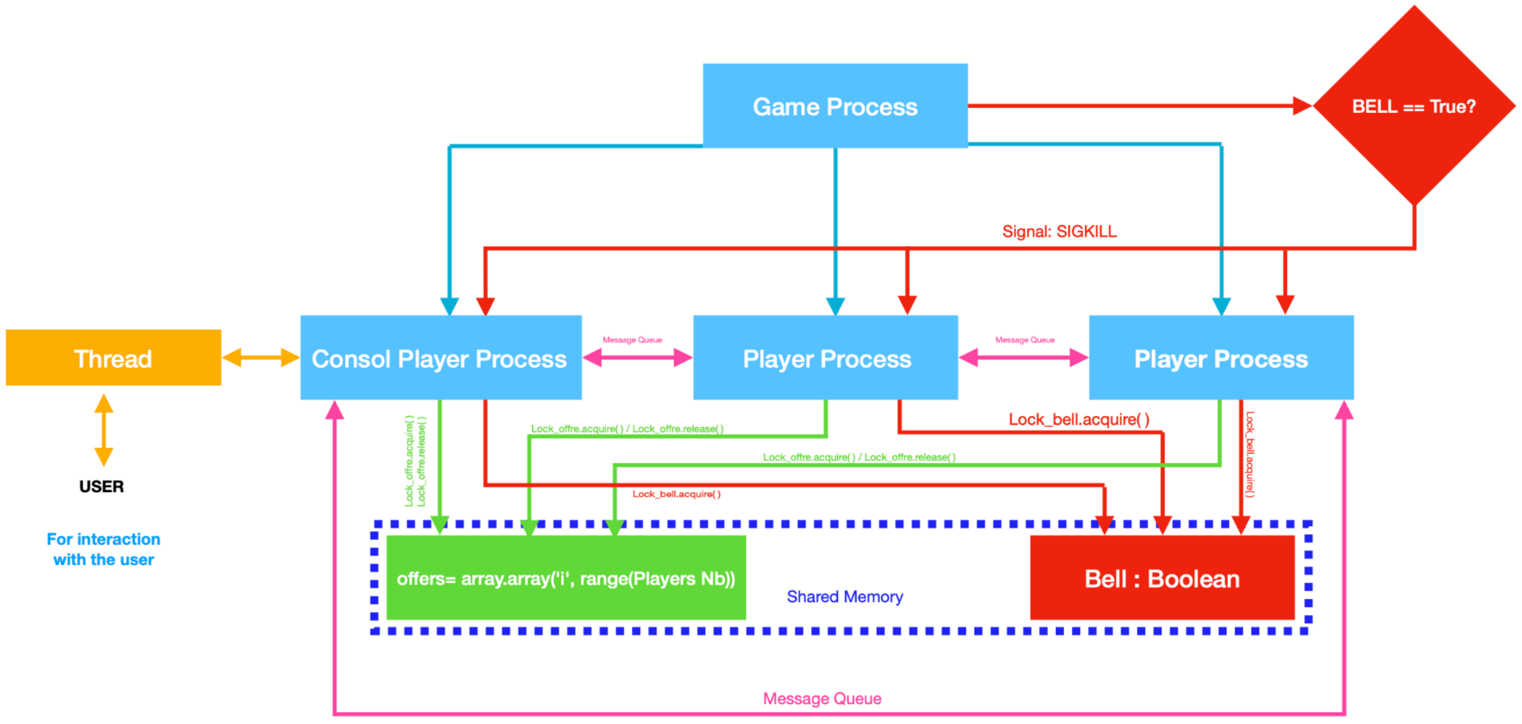
- N= Nb_of_players
- Process_players=[Process(target=Players , args=(offers, Mutex_Bell , Mutex_Player)]

Classes of our Code :

We implemented the cards , the player and the deck of cards as classes with functions as show_hand to display the cards of each player , and function 'deal' to distribute the cards, and function 'discard' to cancel the hand of the players at the end of the game .

```
Class Card(object):

    def __init__(self, suit):#suit for the type of card(train , bike ,airplane …)
        self.suit = suit

    def show(self):

        print("{} ".format(self.suit))

class Deck(object):

    def __init__(self):
        self.cards = []
        self.build()

    def build(self):
        self.cards = []
        for suit in ['bike', 'airplane', 'train', 'car', 'shoes']:
            for val in range(1, 6):
                self.cards.append(Card(suit, val))

    def show(self):
        for c in self.cards:
            c.show()

    # Shuffle the deck
    def shuffle(self, num=1):
        length = len(self.cards)
        for _ in range(num):
            # This is the fisher yates shuffle algorithm
            for i in range(length-1, 0, -1):
                randi = random.randint(0, i)
                if i == randi:
                    continue
                self.cards[i], self.cards[randi] = self.cards[randi], self.cards[i]
            # We can also use the build in shuffle method
            # random.shuffle(self.cards)

    def deal(self):
          return self.cards.pop()

class Player(object):
    def __init__(self, name):
        self.name = name
        self.hand = []

    def draw(self, deck, num=1):
    # Draw n number of cards from a deck, Returns true in n cards are drawn, false if less than that
        for _ in range(num):
            card = deck.deal()
            if card:
                self.hand.append(card)
            else:
                return False
        return True
   # Display all the cards in the players hand
    def show Hand(self):
        print ("{}'s hand: {}".format(self.name, self.hand))
          return self
    def discard(self):
              return self.hand.pop()
```

Implementation diagram :


Game Process
BELL == True?
Signal: SIGKILL
Thread
Consol Player Process
Message Queue
Player Process
Message Queue
Player Process
USER
For interaction with the user
Lock_offre.acquire()
Lock_offre.release()
Lock_offre.acquire() / Lock_offre.release()
Lock_bell.acquire()
Lock_offre.acquire() / Lock_offre.release()
Lock_bell.acquire()
Lock_bell.acquire()
offers= array.array('i', range(Players Nb))
Shared Memory
Bell : Boolean
Message Queue