**PPC project Hannabis**

1. **Conception**

**Structure of program**

* Game Process

It manages the game state, such as card stacks, scores, information tokens and fuse tokens.

* Multiprocessing: There are 2 parallel processes in Game Process, they are used to handle the communication with 2 player processes.
* 2 Player Processes

Each player is a process, these processes are equal, communicating with the game process via a socket, and each player process communicates with each other via a queue by using module “sysv\_ipc”.

**Explanations of choices**

**Parallel Processing**: The game process and the two player processes run in parallel. This design allows the game to process the actions of both players at the same time, improving the overall efficiency of the game.

**Modularity and decoupling**: Each process is responsible for a different task. The game process is responsible for managing the game state, while the player processes are responsible for handling the communication with the players. This division of task makes the code more modular and easier to maintain and extend.

**Communication isolation**: Communication isolation is achieved by using different communication methods, sockets between game processes and player processes, and queues in the sysv\_ipc module between player processes. This helps reduce code complexity and potential communication conflicts.

**Parallel actions of players**: Two player processes can process their own actions at the same time without being affected by the other. This helps to increase the concurrency of the game, so that communication between players does not block each other.

**Flexibility**: By using different communication methods, it can be easier to expand or modify the system in the future, for example, if we need to change the communication method or add more players. This enhances the flexibility of the system.

1. **Implementation plan**

**Important pseudo-code**

**Game Process:**

initialiseGame( )

while not gameOver:

**if** current\_player == player\_id:

notify\_player\_to\_play\_card(conn1)

action = receive\_player\_action(conn1)

**if** action is valid:

process\_play\_card\_action(action)

update\_players\_information( )

switch\_player\_connections(conn1, conn2)

**else**:

notify\_player\_to\_give\_info(conn2)

action = receive\_player\_action(conn2)

if action is valid:

process\_give\_info\_action(action)

update\_players\_information()

# Game over, send results to all players

send\_game\_results(conn1, conn2, is\_game\_win( ))

**Player Process:**

# Thread1 - Game Logic

Thread1\_gameLogic:

while not gameOver:

action = getUserInput( )

validateAction(action)

sendActionToGameProcess(action)

update = getUpdateFromGameProcess( )

displayUpdate(update)

# Thread2 - Communication with Game Process

Thread2\_communicationGame:

while not gameOver:

message = receive.socket( )

processGameMessage(message)

# Thread3 - Communication with Other Players

Thread3\_communicationPlayer:

while not gameOver:

playerMessage = receive.queue( )

processPlayerMessage(playerMessage)

# Pseudo code for handling server socket

try:

while True:

# Receive parsed response from the server

parsed\_response = receive\_message(socket)

# Check if the game is over

if 'game\_over' in parsed\_response:

print\_game\_result(parsed\_response)

break

# Check the required action from the server

if 'action\_required' in parsed\_response:

action\_required = parsed\_response['action\_required']

# Handle the case when the server requests giving information

if action\_required == 'give\_info':

handle\_give\_info\_action()

# Handle the case when the server requests playing a card

if action\_required == 'play\_card':

handle\_play\_card\_action()

except Exception as e:

handle\_error(e)