I – Création du serveur :

Partie 1 : Structures

`thread_args_t`

Structure contenant les arguments pour un thread.

Attributs :

- `server` : pointeur vers la structure `server_s`

```c
typedef struct player_s {
    int id;
    char *buffer;
    int pos_x;
    int pos_y;
    int socket;
    int orientation;
    bool connected;
    int *ressources;
    int level;
    char *look;

    bool is_freeze;
    int freeze_clock;
    int food_clock;
    int last_action;

    int take_object;
    int set_object;
    char *broadcast_text;

    struct player_s *next;
    struct team_s *team;
    struct player_s **players_on_tile;
} player_t;
```

- `buffer` : pointeur vers une chaîne de caractères
- `player` : pointeur vers la structure `player_s`

`server_t`

Structure représentant le serveur.

Attributs :

- `port` : port du serveur
- `client_socket` : tableau de sockets clients
- `address` : structure `sockaddr_in` pour l'adresse
- `master_socket` : socket principale
- `new_socket` : nouvelle socket
- `max_fd` : nombre maximal de descripteurs de fichiers
- `addrlen` : longueur de l'adresse
- `total_resources` : pointeur vers un entier pour le total des ressources
- `game` : pointeur vers la structure `game_s`
- `queue` : pointeur vers la structure `player_s` pour la file d'attente des joueurs

Partie 2 : Fonctions

Fonctions de gestion des sockets

- `create_socket` : crée une socket

```
void create_server(int ac, char **av, server_t *server)
{
    error_handling(ac);
    set_default_parameters(server);
    parse_arguments(ac, av, server);
    memset(server->client_socket, 0, sizeof(server->client_socket));
    create_socket(server);
    bind_socket(server);
    listen_socket(server);
    init_map(server->game);
}
```

La fonction `create_server` est responsable de la création et de l'initialisation d'un serveur. Voici une description rapide des différentes étapes effectuées par la fonction :

- `error_handling(ac)`: Cette fonction vérifie et gère les erreurs éventuelles liées aux arguments de la ligne de commande passés à la fonction `create_server`. Elle garantit que le nombre d'arguments est correct et affiche un message d'erreur approprié en cas de problème.

- `set_default_parameters(server)`: Cette fonction initialise les paramètres par défaut du serveur en affectant des valeurs initiales à la structure `server`. Cela peut inclure l'affectation de valeurs aux membres tels que le port, les sockets, les adresses, etc.
- `parse_arguments(ac, av, server)`: Cette fonction analyse les arguments de la ligne de commande `ac` et `av` et met à jour les paramètres du serveur en conséquence. Elle peut extraire des informations telles que le port à utiliser, les options de configuration, etc., à partir des arguments fournis.
- `memset(server->client_socket, 0, sizeof(server->client_socket))`: Cette instruction initialise le tableau `client_socket` de la structure `server` en le remplissant de zéros. Cela garantit que toutes les sockets clientes sont initialisées à 0, ce qui signifie qu'elles ne sont pas encore connectées.
- `create_socket(server)`: Cette fonction crée une nouvelle socket pour le serveur en utilisant les informations contenues dans la structure `server`. Elle peut utiliser des appels système tels que `socket()` pour créer une nouvelle socket.
- `bind_socket(server)`: Cette fonction lie la socket créée précédemment à une adresse IP et à un port spécifiques. Elle utilise les informations contenues dans la structure `server`, notamment le numéro de port, pour effectuer cette opération.
- `listen_socket(server)`: Cette fonction met la socket en mode d'écoute pour accepter les connexions entrantes. Elle utilise la fonction `listen()` pour spécifier le nombre maximal de connexions en attente autorisées.
- `init_map(server->game)`: Cette fonction initialise la carte de jeu dans la structure `game` du serveur. Elle peut allouer de la mémoire pour la carte, initialiser les cases, les ressources, etc.

Ces différentes étapes permettent de créer et de configurer correctement le serveur, prêt à accepter les connexions des clients et à gérer les interactions du jeu.

- `bind_socket` : associe une adresse à une socket
- `listen_socket` : met la socket en écoute
- `accept_connection` : accepte une nouvelle connexion
- `manage_connection` : gère une connexion existante

```c
int manage_connection(server_t *server)
{
    fd_set read_fds;
    int activity = 0;

    while (1) {
        add_child_socket(server, &read_fds);
        set_select(&read_fds, &activity, server) == -1 ? exit(84) : 0;
        if (FD_ISSET(server->master_socket, &read_fds))
            set_new_socket(server);
        for (int i = 0; i < SOMAXCONN; i++)
            listen_clients(server, i, &read_fds,
            get_player_at_socket(server,
            server->client_socket[i]));
        game_engine(server);
    }
    return (0);
}
```

La boucle principale de la fonction manage_connection gère les connexions et les interactions avec les clients du serveur. Voici une description rapide de son fonctionnement :

- `fd_set read_fds;` : La variable `read_fds` est un ensemble de descripteurs de fichiers utilisé par la fonction `select()` pour surveiller les sockets en lecture.
- `while (1)` : Cette boucle s'exécute en continu car elle utilise une condition `while(1)`, ce qui signifie qu'elle ne se terminera pas tant que le serveur n'est pas arrêté explicitement.
- `add_child_socket(server, &read_fds);` : Cette fonction ajoute les sockets clientes existantes à l'ensemble `read_fds`, afin de les surveiller pour toute activité en lecture.
- `set_select(&read_fds, &activity, server) == -1 ? exit(84) : 0;` : Cette instruction utilise la fonction `select()` pour attendre une activité sur les sockets surveillées. Elle bloque l'exécution jusqu'à ce qu'une activité soit détectée ou jusqu'à ce qu'il y ait une erreur. Si une erreur se produit, le programme est arrêté avec le code de sortie 84.
- `if (FD_ISSET(server->master_socket, &read_fds)) set_new_socket(server);` : Cette condition vérifie si la socket principale du serveur a une activité en attente. Si c'est le cas, cela signifie qu'une nouvelle connexion entrante est en cours, et la fonction `set_new_socket()` est appelée pour accepter cette nouvelle connexion.
- `for (int i = 0; i < SOMAXCONN; i++)` : Cette boucle parcourt toutes les sockets clientes dans `server->client_socket` et effectue des opérations spécifiques pour chaque client.

- `listen_clients(server, i, &read_fds, get_player_at_socket(server, server->client_socket[i]));` : Cette fonction gère les interactions avec un client spécifique identifié par l'indice `i` dans `server->client_socket`. Elle utilise la structure `player_t` associée à cette socket pour traiter les demandes du client.
- `game_engine(server);` : Cette fonction est appelée après avoir géré toutes les connexions clientes dans la boucle. Elle est responsable de la gestion générale du jeu, des mises à jour des joueurs, de la vérification des conditions de victoire, etc.

Une fois que toutes les connexions clientes ont été traitées et que la boucle interne est terminée, la boucle principale reprend son exécution et attend à nouveau les activités sur les sockets.

La fonction `manage_connection` ne retourne pas de valeur significative, elle est généralement utilisée pour maintenir le serveur en cours d'exécution et gérer les connexions clientes de manière continue

- `listen_clients` : écoute les clients

Fonctions de gestion des joueurs

- `get_player_at_socket` : obtient le joueur associé à une socket
- `create_player_back` : crée un nouveau joueur à l'arrière de la liste
- `create_team_back` : crée une nouvelle équipe à l'arrière de la liste
- `create_egg_back` : crée un nouvel œuf à l'arrière de la liste
- `delete_egg_at_id` : supprime un œuf en fonction de son identifiant
- `get_team_at_name` : obtient une équipe en fonction de son nom
- `get_team_number` : obtient le nombre d'équipes

Autres fonctions de gestion

- `free_memory` : libère la mémoire allouée pour le serveur
- `get_freeze_time_cmd` : obtient le temps de congélation d'un joueur
- `parse_arguments` : analyse les arguments de la ligne de commande
- `eof_reach` : atteint la fin du fichier pour un joueur

Partie 3 : Fonctions de gestion des requêtes

Fonctions de gestion des requêtes non connectées

- `handle_not_connected_request` : traite une requête non connectée
- `handle_gui_request` : traite une requête GUI
- `parse_request` : analyse une requête
- `detect_command` : détecte une commande dans une chaîne de caractères

Fonctions de récupération d'informations

- `get_tile` : obtient les informations d'une case
- `get_pin` : obtient les informations d'un joueur (pin)
- `get_ppo` : obtient les informations d'un joueur (ppo)
- `get_map_size` : obtient la taille de la carte
- `get_name` : obtient le nom du serveur
- `get_map` : obtient la carte du jeu
- `get_total_player_number` : obtient le nombre total de joueurs dans le jeu

Partie 4 : Fonctions utilitaires

- `get_tile_by_pos` : obtient une case en fonction de ses coordonnées
- `find_empty_tile` : trouve une case vide dans le jeu
- `calculate_resources_available` : calcule les ressources disponibles dans le jeu
- `get_nb_players_on_tile` : obtient le nombre de joueurs sur une case
- `delete_player_at_socket` : supprime un joueur en fonction de sa socket
- `delete_player_from_tile` : supprime un joueur d'une case
- `eject_eggs` : éjecte les œufs d'une case
- `eject_west` : éjecte un joueur vers l'ouest

Partie 5 : Fonctions de spawn des joueurs

- `spawn_players` : fait apparaître les joueurs
- `spawn_player` : fait apparaître un joueur
- `spawn_all_players` : fait apparaître tous les joueurs

Partie 6 : Fonctions d'affichage

- `print_players` : affiche les joueurs
- `print_teams` : affiche les équipes
- `print_map` : affiche la carte du jeu

Partie 7 : Fonctions de formatage des données

- `format_bct` : formate une case pour la réponse BCT
- `format_pin` : formate un joueur pour la réponse PIN

Partie 8 : Fonctions du moteur de jeu

- `start_game` : lance le jeu
- `game_engine` : moteur de jeu
- `send_to_all_players` : envoie un message à tous les joueurs
- `send_to_all_graphic` : envoie un message à tous les clients graphiques

Partie 9 : Fonctions d'incantation

- Fonctions pour l'incantation
- Fonctions pour l'ancantation

Partie 10 : Fonctions de déplacement

- Fonctions pour les déplacements des joueurs

Partie 11 : Fonctions pour la gestion des actions de congélation

```c
void freeze_action(game_t *game, player_t *player, int action, char *buffer)
{
    if (player->is_freeze == true)
        return;
    player->is_freeze = true;
    switch (action) {
        case INCANTATION:
            incantation_cmd(game, player);
            break;
        case TAKE:
            take_freeze(game, player, buffer);
            break;
        case SET:
            set_freeze(game, player, buffer);
            break;
        case BROADCAST:
            broadcast_freeze(game, player, buffer);
            break;
        default:
            freeze_action_2(game, player, action);
            break;
    }
}
```

```c
void process_action(player_t *player, game_t *game)
{
    if (player->last_action == INCANTATION)
        incantation_succes(player, get_tile_by_pos(game->map, player->pos_x, player->pos_y), ressoources_per_lvl(player->level), game);
    if (player->last_action == FORK)
        action_fork(game, player);
    if (player->last_action == INVENTORY)
        action_inventory(game, player);
    if (player->last_action == FORWARD)
        action_forward(game, player);
    if (player->last_action == RIGHT || player->last_action == LEFT)
        action_turn(game, player);
    if (player->last_action == LOOK)
        look_cmd(game, player);
    if (player->last_action == BROADCAST)
        action_broadcast(game, player);
    if (player->last_action == EJECT)
        action_eject(game, player);
    if (player->last_action == TAKE)
        action_take(game, player);
    if (player->last_action == SET)
        action_set(game, player);
}
```

Le système de "freeze" et "unfreeze" est utilisé pour suspendre temporairement les actions d'un joueur dans le jeu. Voici une explication rapide de son fonctionnement :

La fonction `freeze_action` est appelée lorsque le joueur effectue une action spécifique et qu'il doit être gelé pendant un certain temps. Elle prend en compte le type d'action effectuée (`action`) et le joueur concerné (`player`), ainsi que d'autres informations éventuelles (`buffer`).

1. Tout d'abord, la fonction vérifie si le joueur est déjà gelé (`player->is_freeze == true`). Si c'est le cas, la fonction se termine immédiatement sans rien faire.
2. Si le joueur n'est pas déjà gelé, la variable `player->is_freeze` est définie sur `true`, indiquant que le joueur est gelé.
3. Ensuite, un `switch` est utilisé pour traiter différentes actions (`action`) qui peuvent être gelées. Selon le type d'action, une fonction spécifique est appelée pour gérer cette action en mode gelé. Par exemple, si l'action est `INCANTATION`, la fonction `incantation_cmd` est appelée pour traiter l'incantation en mode gelé.
4. Si l'action ne correspond à aucun des cas spécifiques du `switch`, la fonction `freeze_action_2` est appelée pour effectuer des actions supplémentaires spécifiques à d'autres types d'action.

Le but de cette fonction est de suspendre temporairement les actions du joueur lorsque celui-ci est gelé, en lui empêchant d'exécuter certaines actions spécifiques. Une fois que le joueur est gelé, ses actions seront gérées différemment jusqu'à ce qu'il soit "unfreezé".

La fonction `process_action` est appelée lorsque le joueur doit exécuter l'action qui a été gelée une fois que le joueur est "unfreezé". Elle vérifie la dernière action effectuée par le joueur (`player->last_action`) et appelle la fonction correspondante pour traiter cette action.

Chaque action spécifique (par exemple, `INCANTATION`, `FORWARD`, `TAKE`, `SET`, etc.) a une fonction associée qui effectue les opérations appropriées dans le jeu. En fonction de l'action précédente du joueur, la fonction correspondante est appelée pour reprendre le traitement de cette action une fois que le joueur est "unfreezé".

Cela permet de gérer le système de gel et de reprise des actions des joueurs dans le jeu en fonction de leur état de gel.

- `freeze_action` : action de congélation
- `take_freeze` : prendre congélation
- `set_freeze` : définir congélation
- `check_for_freeze` : vérifier la congélation
- `broadcast_freeze` : diffuser la congélation
- `action_forward` : action d'avancer
- `action_turn` : action de tourner

- `action_take` : action de prendre
- `action_set` : action de déposer
- `action_eject` : action d'éjection
- `action_inventory` : action d'inventaire
- `action_fork` : action de fork
- `action_broadcast` : action de diffusion

Partie 12 : Fonctions utilitaires pour la gestion des ressources

- Fonctions pour l'obtention des noms des ressources

Partie 13 : Fonctions de visualisation

- `get_name_ressources_on_tile` : obtient les noms des ressources sur une case
- `get_name_player_on_tile` : obtient le nom des joueurs sur une case
- `look_cmd` : commande "voir" (look)

Partie 14 : Fonctions utilitaires

- `min` : renvoie la valeur minimale entre deux entiers