Bardinet Romain

RAPPORT : Système d'Exploitation 2

DEDALUS

Sommaire :

1/ Introduction

Dedalus est un jeu dans lequel des joueurs sont enfermés dans un labyrinthe donné, en présence de minotaures. Le but des joueurs et de survivre contre les minotaures pour éventuellement s'échapper et accéder au niveau suivant. Ils ont pour cela un temps limité.

Dans ce projet, les joueurs sont dirigés par des intelligences artificielles. Notre objectif est d'optimiser le fonctionnement du jeu, en réduisant le nombre de paramètres à entrer pour lancer une partie depuis le terminal, puis en modifiant l'architecture du jeu de manière à pouvoir traiter le comportement de plusieurs joueurs en même temps.
Le développement suivit se divise en 3 parties : Dans une première partie nous faisons en sorte que l'utilisateur n'ai plus à entrer les pid des fenêtres dans lesquelles il veut afficher la vue de chaque joueur. Ce traitement se fait automatiquement à partir du nombre de joueurs présents dans le jeu. Dans une seconde partie, nous modifions la fonction qui prend en charge les mouvements des joueurs que retourne chaque IA de manière à ce que toutes les IA puissent calculer un mouvement en même temps, utilisant ainsi les ressources disponibles plus efficacement. Enfin, nous ferons en sorte que l'appel des IA se fasse de manière continue, ainsi, dès qu'une IA a fini son calcul de mouvement, la carte et les affichages du jeu sont actualisés.

2/ Développement

Partie 1 : Gestion de l'affichage

Dans cette partie, on veut ouvrir un terminal XTERM par joueur présent sur la carte au début de la partie.
Pour cela on doit :

- Obtenir nbPlayer, le nombre de joueurs présents sur la carte

- Ouvrir nbPlayer fois un terminal XTERM, de manière espacée sur l'écran
- Récupérer le PID correct de chaque terminal
- Ajouter chaque PID dans la liste des PID de processus auxquels seront envoyé le données à afficher pour chaque joueur.
- Assurer la fermeture des terminaux XTERM, par la terminaison de leur processus.

On utilisera :

La fonction execl(const char *path, const char *arg, ...), pour ouvrir les terminaux.

La fonction kill(pid_t pid), pour fermer les terminaux.

La fonction sprintf(string,string,parameters) pour implémenter une chaine de caractères variable à fournir à execl.

La fonction fork() qui nous permet de faire appel à execl sans interrompre le programme principal.

On choisit de modifier game_init (dans game.c), fonction appelée pour initialiser le jeu.
En effet cette fonction contient une boucle for qui initialise différents paramètres pour chaque joueur. On va donc ouvrir un terminal à chaque itération de la boucle, comme ci :

```
for (size_t i = 0; i < pGame->nbPlayer; ++i) {

  // Get the display to use with this player.
  pid_t display;
  pid_t forkpid = fork();
  if(forkpid==0){
    char str[20];
    sprintf(str,"60x20+%d+0",(int)i*450);
    execl("/usr/bin/xterm","xterm","-geometry",str,NULL);
  }else{
    display = _character_get_child_pid(forkpid);
    pid_t display_window = forkpid;
    //Updating pConfig aswell
    config_add_display(pConf,display_window);
  }

  // Init player
  pGame->playerA[i] =
      character_init(PLAYER, display, false, "Theseus", pDefHealth, pAi);
  pGame->playerA[i].pos = pAPos[i];
  pGame->playerA[i].pMask = map_mask_init(pMap);
  map_mask_add(pGame->playerA[i].pMask, pAPos[i]);
  // NOTE: Targets are not set yet
}
```

A chaque itération (autant d'itération que de joueur à afficher), on fait appel à fork() pour dupliquer le processus. La boucle if(forkpid==0) n'est lue que par le fils. Par soucis de présentation du jeu à l'utilisateur, chaque fenêtre ouverte est décalée par rapport à la précédente pour qu'il n'y ai pas de chevauchements. On fait appel à sprintf pour remplir une chaîne de caractères en fonction de l'indice "i" de parcours de boucle. La fonction execl

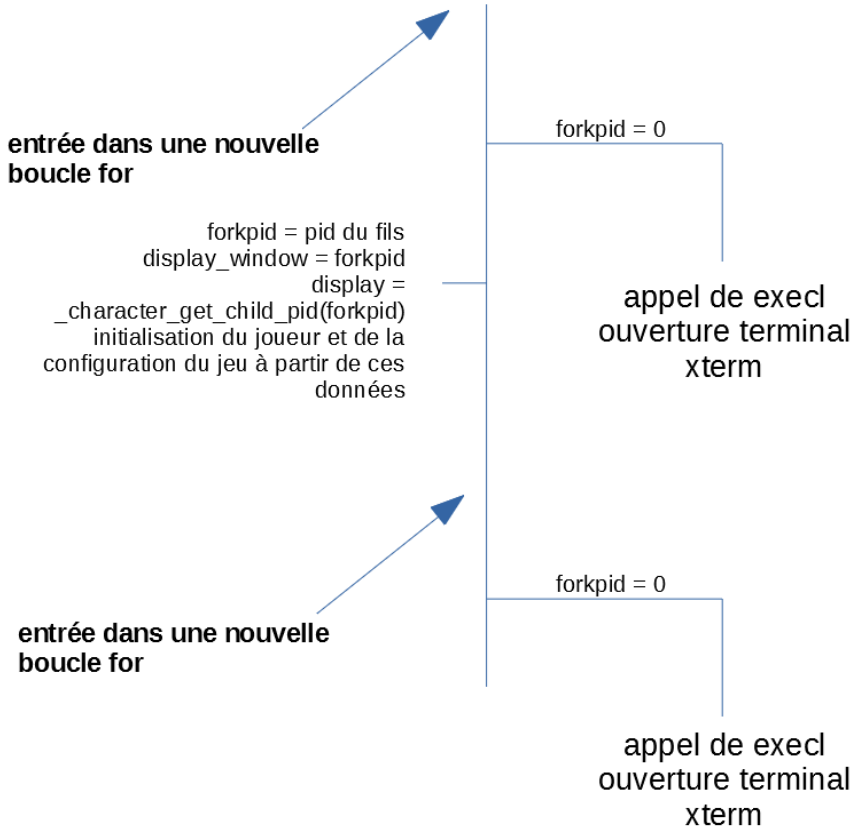
remplace la suite du code par ce que l'on entre en commande. Ainsi le fils n'as pas besoin d'exit(status) en fin de boucle.

Depuis le processus père, on veut maintenant :

- le pid à envoyer à character_init, qui doit être celui du bash (display)
- le pid à stocker dans la liste des pid dans lesquels écrire, qui doit être celui de xterm (display_window)

Le PID du terminal xterm est celui obtenu lors du fork en début de boucle for. On a juste à copier la valeur de forkpid. Pour obtenir le PID du bash dans lequel s'affiche les données du jeu, on utilise la fonction _character_get_child_pid(pid) sur le PID du terminal xterm lui même.
La partie //Init player se charge de finir l'initialisation de chaque joueur.

Arbre de processus :



Il faut maintenant fermer chacun des processus créés.
Pour cela une simple boucle for itérant sur le nombre de joueur est requise dans les étapes de libération de mémoire à la fin du main() de dedalus.c.

```
// Clear at the end

for(int k = 0;k<(int)game.nbPlayer;k++){
  kill(config.displayPidA[k],SIGTERM);
}

game_delete(&game);
config_delete(&config);

return EXIT_SUCCESS;
```

On retrouve le PID de chaque terminal xterm à terminer dans la structure “config”, en effet on les avait stocké ici lors de l’initialisation des joueurs avec la fonction config_add_display.

La fonction kill assure leur suppression, SIGTERM est le signal par défaut à donner en paramètre de kill() lors de la terminaison d’un signal.

On a donc à présent des terminaux xterm qui s’ouvrent en fonction du nombre de joueurs présents dans le jeu, et qui se ferment quand la partie s'arrête.

Partie 2 : Parallélisation du calcul de mouvement par les IA.

Dans cette partie on veut paralléliser le calcul de chaque IA. A chaque nouvelle étape du jeu, toutes les IA sont sollicitées en même temps et renvoient ensuite le résultat de leur calcul au processus père qui se charge de les envoyer au moteur de jeu.

Les IA sont appelées initialement par une boucle for qui les appelle tour à tour via la fonction _game_get_moves_propositions(gamet*pGame,moves_prop_t*pMoves,size_t nbChar). On peut ré-utiliser cette boucle pour implémenter un système de parallélisation :
Le processus père parcourt la boucle, à chaque itération, il fork un processus fils qui est terminé (avec exit(entier)) avant la fin de la boucle. Chaque processus fils renvoie une direction dir (entre 0 et 9), l’index de la boucle i (il est important car il représente l’identifiant du joueur, nécessaire pour remplir *correctement* le tableau pMoves par la suite), et le status de .cheated boolcheck (soit 1 soit 0).

La mémoire est dupliquée quand on appel fork(), de ce fait, un processus fils n’a pas accès à la mémoire du père. Il ne peut communiquer le résultat des calculs de l’IA qu’au travers de sa valeur de retour.

Voici donc le procédé pour coder ces 3 informations dans un entier :

- L’index est multiplié par 20
- On ajoute 10 si pMovies[index].cheated == true
- On ajoute enfin la valeur de la direction
-

Cette somme que nommée “final” dans le code est envoyée depuis le fils en écrivant exit(final).

On obtient donc le procédé de parallélisation suivant:

```
for(i=0;i<nbChar;++i){
  pid_t pidf = fork();
  if(pidf==0){

    int final;
    character_t* pC = pMoves[i].c;
    int dir = (int)character_propose_move(pC, pGame->pMap, &(pMoves[i].cheated));
    int boolcheck=0;

    if(pMoves[i].cheated==true){
      boolcheck+=10;
    }

    final = 20*(int)i + boolcheck + dir;
    exit(final);
  }
}
```

Processus de récupération des valeurs envoyées par les fils :

Nous allons utiliser ici wait(&status) de la bibliothèque <sys/wait.h> qui met à jour status dès qu'elle reçoit le message d'exit d'un des fils du père. Les processus fils se terminent en fonctions de la vitesse de l'IA utilisée, celle-ci diffère potentiellement à chaque boucle, il n'est donc pas possible de savoir de quel processus on reçoit le signal exit. C'est en fait pour cela qu'on a codé index dans la valeur de retour de ce dernier.

On implémente donc une boucle while avec un compteur "w" valant 0 au début, incrémenté de 1 à chaque boucle. A chaque boucle on utilise wait(&status) pour attendre le retour d'un fils et traiter la valeur reçue.

La condition d'arrêt de la boucle est w<nbChar, à ce point, tous les fils ont retourné une valeur.

La valeur prise par status après un appel de wait(&status) par le père est utilisable via WEXITSTATUS(status) qui renvoie la valeur "final", valeur que l'on nomme "output" dans le père.

Pour obtenir les 3 valeurs dir, i et boolcheck, on procède ainsi :

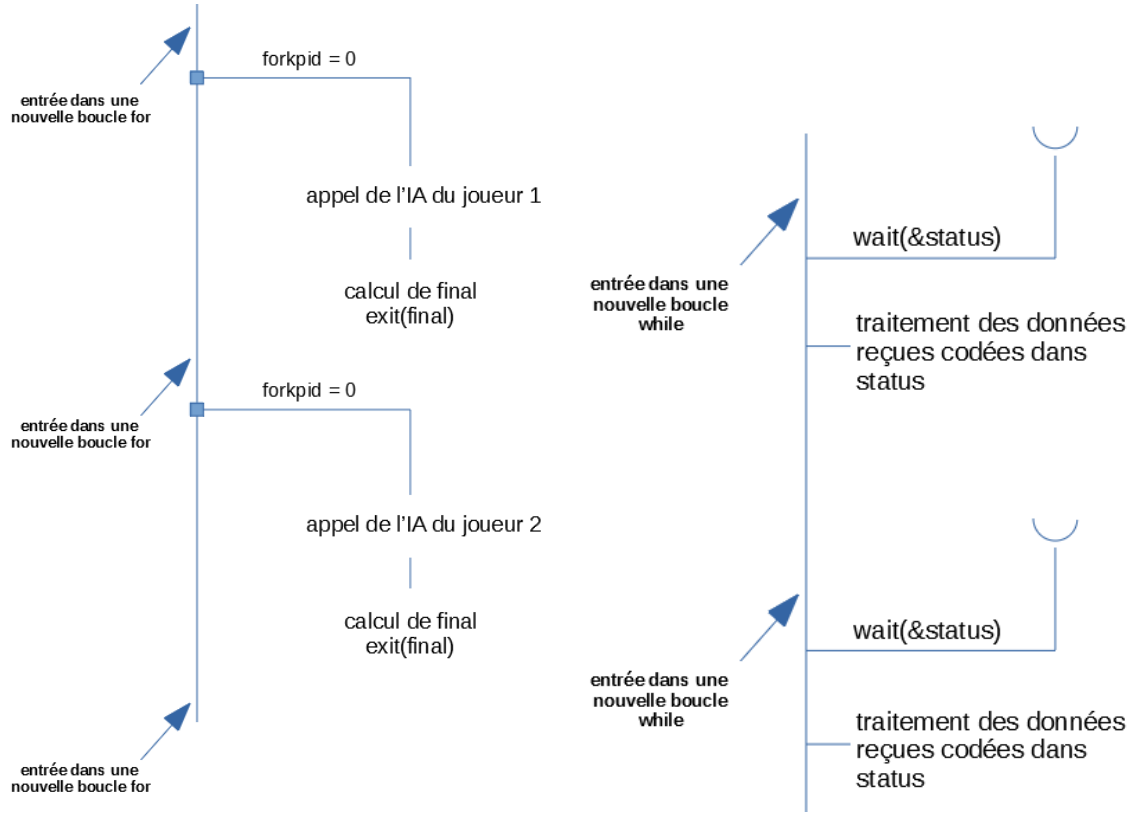
- on divise (division entière) output par 20 pour obtenir l'index
- si output%20 est supérieur à 10, alors c'est un mouvement illégal, cheatedstatus = 1, sinon, il reste à 0.
- la direction calculée par l'IA est output%20 - cheatedstatus*10

Il suffit ensuite de mettre à jour le tableau pMoves.

Implémentation d'une boucle while pour récupérer les valeurs envoyées par les fils :

```
int w = 0;
while(w<(int)nbChar){

  wait(&status);
  int output = WEXITSTATUS(status);
  int index = output/20;
  int cheatedstatus;
  if(output%20<10){
    cheatedstatus = 0;
    pMoves[index].cheated = false;
  }else{
    cheatedstatus = 1;
    pMoves[index].cheated = true;
  }

  int dir = output%20 - 10*cheatedstatus;

  pMoves[index].move = dir;

  w++;
}
```

L'arbre des processus suit le modèle suivant :



Notes sur le comportement de l'IA si on commente la ligne 44 du fichier ai_random.c :

```
srand(time(NULL) * getpid() * rand());
```

On constate que les joueurs concernés ne font que des allers-retours, l'aspect aléatoir de leur comportement est perdu. En effet, il faut mettre à jour la "seed" de la fonction srand, sinon elle envoie toujours la même valeur.

Partie 3 : Fonctionnement asynchrone

L'idée de cette partie est de questionner les IA de manière continue, dès qu'une IA a fini de calculer un mouvement à jouer, elle se met à calculer le suivant. Et ce jusqu'à ce qu'il n'y ait plus de joueurs vivants sur le map. Nous allons utiliser la boucle while(pGame->nbPlayerOnBoard>0) déjà présente dans la fonction game_start du fichier game.c. C'est en effet la boucle principale du moteur du jeu, puisqu'elle gère le mouvement de chaque joueur ainsi que les vérifications obligatoire à l'implémentation de chaque déplacement, puis le rafraîchissement des terminaux d'affichage avec la nouvelle carte des positions.

Nous procéderons de la manière suivante : avant d'entrer dans la boucle, un tableau plPM est initialisé via un malloc, il contient les informations nécessaires au traçage de chaque joueur :

```
typedef struct pid_move {
  pid_t pid;   ///< Carries the pid of the player
  int move;    ///< Carries the move to be played
  bool update; ///< true if no AI is currently computing a new move, false otherwise
} pid_move_t;
```

Il faut l'initialiser en mettant toutes les valeurs de "update" sur vrai (à voir comme "une mise à jour de movement est demandée pour ce joueur").

On entre ensuite dans la boucle while(pGame->nbPlayerOnBoard>0).

Création / Réanimation des fils :

On entame le processus de parallélisation avec une boucle for(int i=0;i<(int)nbChar;i++), un fils par joueur.

Pour chaque index (chaque joueur) on vérifie à l'aide de plPM qu'une IA ne soit pas déjà entrain de calculer un mouvement à venir, auquel cas, aucun appel n'est fait à l'IA du joueur et on passe au joueur suivant.

On duplique le processus. Le processus fils fait appel à la fonction character_propose_move, celle-ci fait elle même appel à l'IA du joueur considéré.

On conserve le même fonctionnement que dans la partie précédente pour le transfert de données entre le fils et le père.
Le processus père entre le PID du processus entrain de calculer le mouvement prochain à l'indice correspondant dans le tableau pIPM.

On obtient donc le code suivant pour cette première sous-partie :

```
for(int i = 0; i<(int)nbChar ; i++){

  if(pIPM[i].update == true){
    //Checking whether player needs to be update or if a process is already running

    pid_t pid_f = fork();

    if(pid_f==0){

      int final;
      character_t* pC = moves[i].c;
      int dir = (int)character_propose_move(pC, pGame->pMap, &(moves[i].cheated));
      int boolcheck=0;

      if(moves[i].cheated==true){
        boolcheck+=10;
      }
      final = 20*(int)i + boolcheck + dir +1;
      exit(final);

    }else{ //Adding the PID of the son in pIPM to check whether it is still running later
      pIPM[i].pid = pid_f;
    }

  }
}
```

Acquisition des données envoyées par les fils et actualisation du jeu :

On implémente une boucle for(int k = 0; k<(int)nbChar;k++) qui va scanner la valeur de retour de chaque processus et modifier les paramètres du jeu en conséquence.
Contrairement à la partie précédente, on s'attend ici à ce que certains processus fils n'aient pas fini et dans ce cas on s'intéresse aux autres.
Cela se fait à l'aide du signal "WNOHANG" passé en paramètre de waitpid :

```
for(int k = 0; k<(int)nbChar;k++){

  pid_t output_raw = waitpid(pIPM[k].pid,&status,WNOHANG);
  int output = WEXITSTATUS(status);

  //WNOHANG so that if the process hasnt finished, we're looking for another
  //pid of a process that might have exited already
  if(output_raw!=0){ //if the scanned pid has finished and has returned a value
```

On regarde dans pIPM à l'index k, on récupère le PID. C'est le PID du processus fils qui calcul le mouvement du joueur n°k. Si celui-ci n'a pas fini (pas atteint exit("final")) alors

output_raw = 0. Dans ce cas, on s'assure que pIPM[k].update = false pour ne pas dupliquer un autre processus de calcul, et on passe à un autre joueur :

```
}else{
  pIPM[k].update = false;
  //waitpid returned 0 therefore the process hasnt exited yet, no need to fork (update) again
```

Dans le cas ou le PID scanné est celui d'un processus fils qui a fini ses calculs, on peut mettre à jour le jeu avec le nouveau mouvement reçu.

De la même manière que dans partie 2, on récupère l'index, la direction, et le cheatedstatus à partir de output.

Avant de procéder aux modifications des données du jeu, on vérifie que le joueur n'a pas été tué durant le calcul de son prochain mouvement. Pour cela on regarde son statut dans moves[index].c->type. Si celui-ci est DEAD, alors on ne fait rien et on passe au joueur suivant.

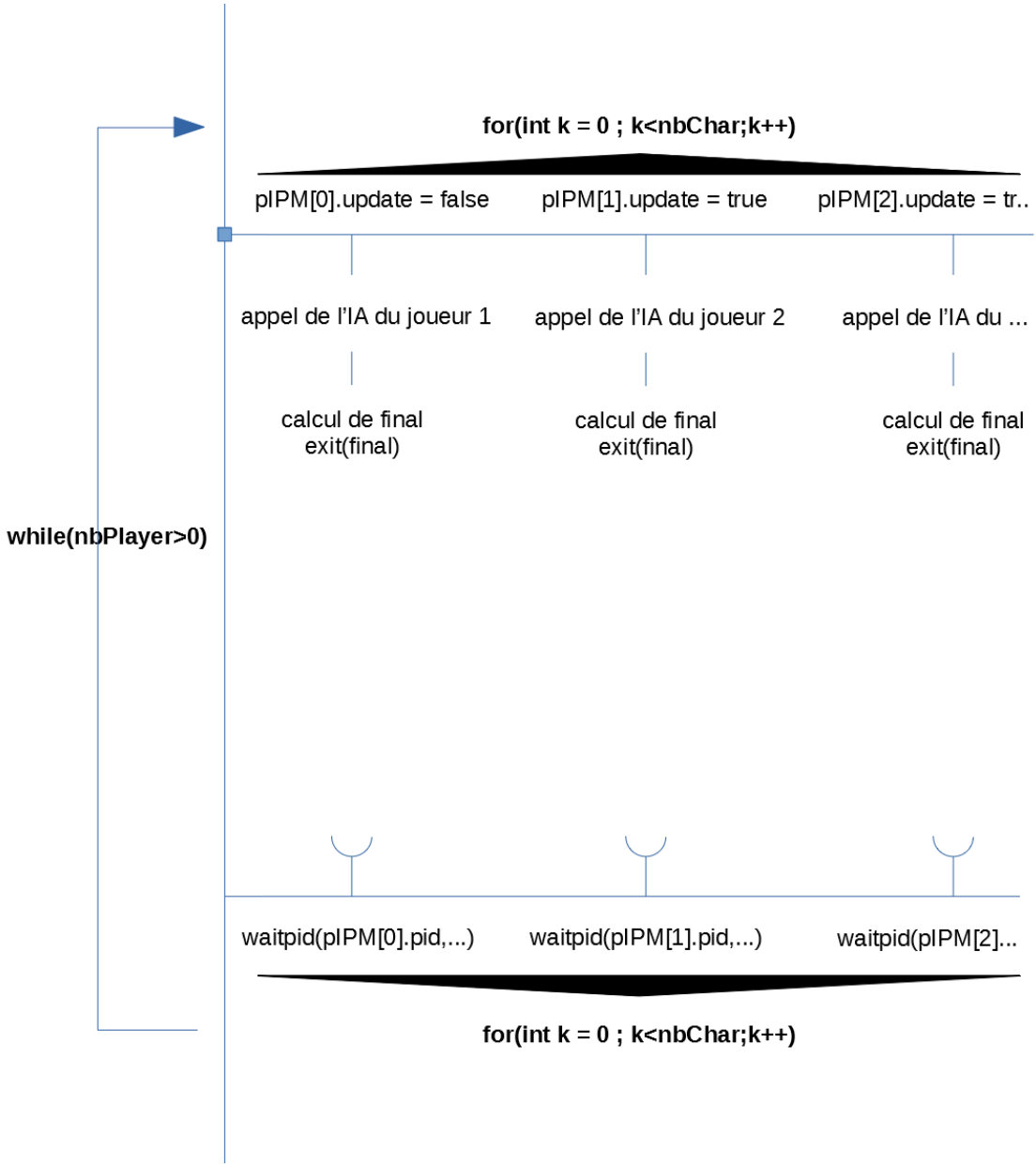
```
//Making sure the character is still alive
if(!(moves[index].c->type==DEAD)){
```

Dans le cas où le joueur est toujours vivant, on vérifie que sa future position soit toujours valide (il se peut que pendant le temps de calcul, un autre joueur ou élément ait pris cette place), on fait cela à l'aide de la fonction _character_can_go. Si la position a en effet été prise entre temps, on ordonne au joueur de rester sur place. Sinon, on fait jouer le joueur et on met à jour l'affichage.

```
moves[index].move=dir;

//Checking whether the movement is directing the player toward a valid position or not
if(_character_can_go(*(pGame->pMap),moves[index].c->pos,dir)){

  // Play character
  _game_play_character(pGame, moves[index].c,moves[index].move,moves[index].cheated);

  // refresh displays
  _game_play_refresh_ui(pGame);

  // Fights
  _game_fight_manager(pGame);

}else{ //The next position was already taken, the player is asked to stand still
  moves[index].move=0;
}
```

Si un processus fils a renvoyé un mouvement, il faut en dupliquer un autre lors de la prochaine boucle while(pGame->nbPlayerOnBoard>0), pour cela, il faut mettre à jour pIPM[k].update sur true.

Finalement on obtient l'arbre de processus suivant :



3/ CONCLUSION :

Dans ce projet, nous avons étudié la mise en parallélisation de différents calcul, à différentes fins. Pour pouvoir lancer d'autres tâches sans interrompre le programme principal, ou simplement dans le but de gagner en vitesse de calcul quand plusieurs calculs peuvent être réalisés en même temps. Si nous n'avons pas dressé de graphique montrant le nombre de positions attribuées par rapport au temps écoulé depuis le lancement du jeu, on constate tout de même une nette différence en terme de vitesse de déroulement du jeu entre le code implémenté dans la partie 1 (pas de parallélisation des calculs de mouvement) et le code implémenté en partie 3 (comportement asynchrone des joueurs). Une étude plus poussée des IA employées aurait pu montrer l'intérêt d'un comportement asynchrone des joueurs, en comptant et comparant par exemple le nombre de fois qu'un joueur "intelligent" se déplace par rapport à un joueur "aléatoire".