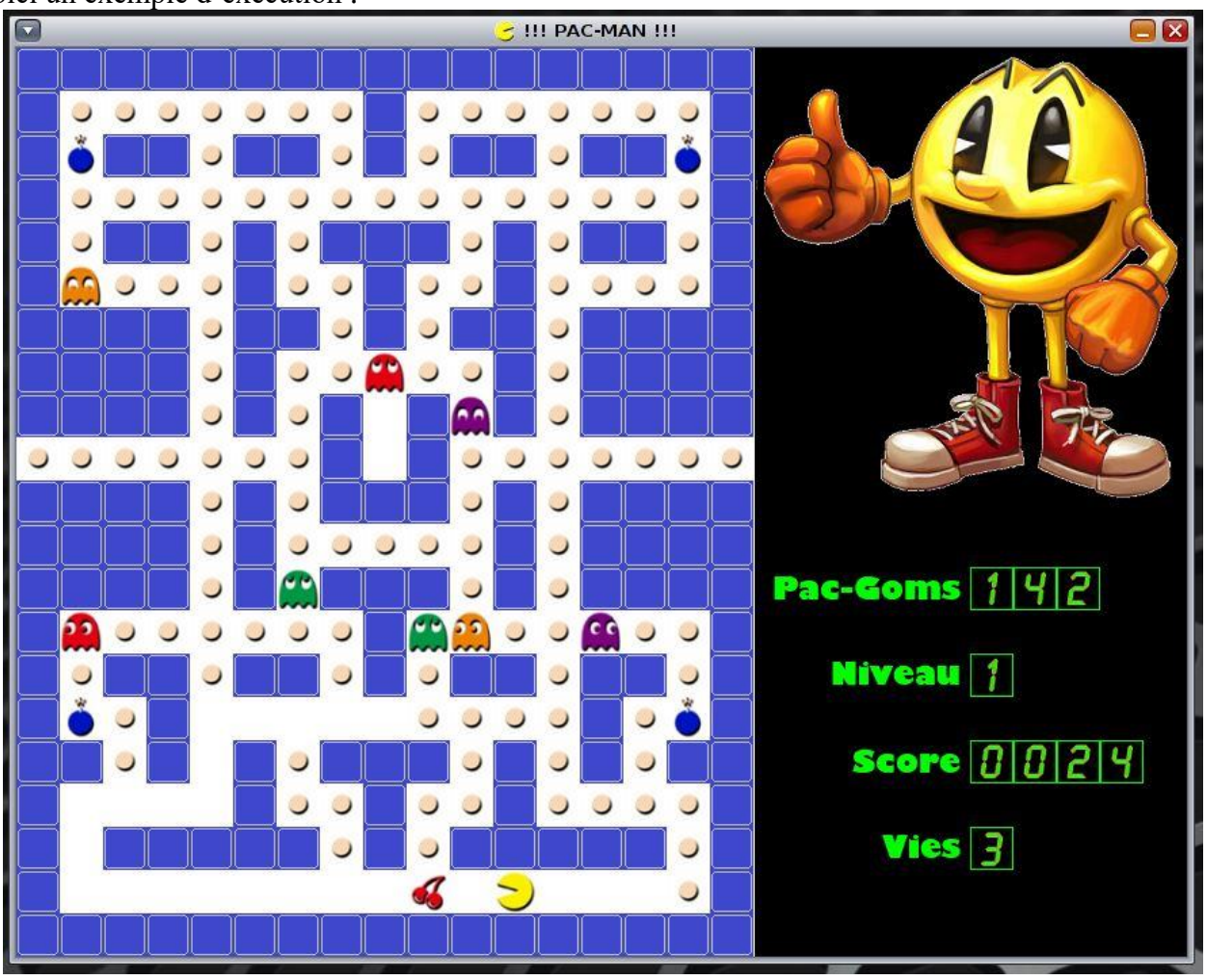
Laboratoire de Multi-Threading - Enoncé du dossier final
Année académique 2024-2025
Jeu de « Pac-Man »

Il s'agit de créer un jeu du type « Pac-Man » pour un joueur. Quelques rappels sur ce jeu déjà bien connu... Le joueur est responsable du déplacement de Pac-Man dans la grille de jeu. Sans intervention du joueur, Pac-Man avance droit devant lui jusqu'à ce qu'il rencontre un des murs du labyrinthe (ou un fantôme, dans ce cas, il meurt…). Ses changements de direction sont imposés à l'aide du clavier. Le but de Pac-Man est de manger tous les « pac-goms » qui se trouvent dans la grille du jeu, et cela sans se faire attraper par les fantômes qui se baladent, eux aussi, dans le labyrinthe. Au cours d'une partie, le joueur dispose de 3 vies. Dès que le joueur a perdu ses 3 vies, la partie se termine. La grille de jeu contient, au départ, 4 « super pac-goms » (les bleus) qui, une fois avalés par Pac-Man, rendent les fantômes « comestibles » par Pac-Man pendant un certain nombre de secondes aléatoire. Une fois tous les pac-goms disparus, le niveau du jeu change, ce qui se traduit simplement par une accélération de Pac-Man et des fantômes, ainsi que le remplissage de la grille de jeu par de nouveaux pac-goms. Enfin, un bonus (une paire de cerises) apparaît de manière aléatoire dans une zone libre de la grille de jeu. Si Pac-Man parvient à dévorer ce bonus avant qu'il ne disparaisse, cela lui octroie des points supplémentaires au score.

Voici un exemple d'exécution :



Notez dès à présent que plusieurs librairies vous sont fournies
(lien GitHub : https://github.com/hepl-dsoo/LaboThread2025_Enonce) :

- **GrilleSDL** : librairie graphique, basée sur SDL, qui permet de gérer une grille dans la fenêtre graphique à la manière d'un simple tableau à 2 dimensions. Elle permet de dessiner, dans une case déterminée de la grille, différents « sprites » (obtenus à partir d'images bitmap fournies)
- **images :** répertoire contenant toutes les images bitmap nécessaires à l'application : image de fond, pacman, pacgoms, mur, fantômes, …
- **Ressources :** Module permettant de charger les ressources graphiques de l'application, de définir un certain nombre de macros associées aux sprites propres à l'application et des fonctions permettant d'afficher des sprites précis (pacgom, mur, …) dans la fenêtre graphique.

De plus, vous trouverez le fichier **PacMan.cpp** qui contient déjà les bases de votre application (dessin de la grille de jeu) et dans lequel vous verrez des exemples d'utilisation de la librairie GrilleSDL et du module Ressources. Vous ne devez donc en aucune façon programmer la moindre fonction qui a un lien avec la fenêtre graphique. Vous ne devrez accéder à la fenêtre de jeu que via les fonctions de la librairie GrilleSDL et du module Ressources. Vous devez donc vous concentrer uniquement sur la programmation des threads !

Les choses étant dites, venons-en aux détails de l'application… La grille de jeu sera représentée par un tableau à 2 dimensions (variable **tab**) défini en global. Il s'agit d'un tableau de structures du type

```
typedef struct {
  int presence;
  pthread_t tid;
} S_CASE;
```

où **presence** correspond à l'entité présente à cette case et peut prendre une des valeurs suivantes (macros définies dans PacMan.cpp) :

- 0 : VIDE
- 1 : MUR
- 2 : PACMAN
- 3 : PACGOM
- 4 : SUPERPACGOM
- 5 : BONUS (les cerises)
- 6 : FANTOME

Dans certains cas, une entité est gérée par un thread (PACMAN et FANTOME). Dans ces cas, **tid** représente l'identifiant du thread (pthread_t) correspondant.

Le tableau tab et la librairie graphique fournie sont totalement indépendants. Dès lors, si le thread Pac-Man veut se placer à la case (7,3), c'est-à-dire à la ligne 7 et la colonne 3, vous devrez coder :

```
setTab(7,3,PACMAN,pthread_self());  // gère la logique de tout le jeu
DessinePacMan(7,3,DROITE) ;  // fonction du module Ressources,
                             // pour dessiner dans la fenêtre graphique
```

où **setTab()** est une fonction fournie qui permet de modifier le tableau tab. Pour effacer Pac-Man, il faut coder

```
setTab(7,3); // gère la logique de tout le jeu (remet presence et tid à 0)
EffaceCarre(7,3) ;    // fonction du module GrilleSDL,
                      // pour effacer une case dans la fenêtre graphique
```

Afin de réaliser cette application, il vous est demandé de suivre les étapes suivantes dans l'ordre et de **respecter les contraintes d'implémentation citées**, **même si elles ne vous paraissent pas les plus appropriées** (le but étant d'apprendre les techniques des threads et non d'apprendre à faire un jeu 😉).

Etape 1 : Création du thread Pac-Man

Dans un premier temps, le thread principal ne lancera qu'un seul thread, le threadPacMan, et se synchronisera dessus par un pthread_join avant de terminer le processus. La position et la direction dans laquelle Pac-Man se dirige sont caractérisées par les variables globales suivantes :

- deux entiers **L** et **C** qui représentent sa ligne et sa colonne dans la grille de jeu,
- un entier **dir** représentant sa direction à un moment donné. La variable dir peut donc avoir une des valeurs suivantes : HAUT, BAS, GAUCHE, DROITE qui sont des macros qui sont fournies dans le fichier Ressources.h

Au départ, Pac-Man devra apparaître sur la case (15,8), c'est-à-dire ligne L=15, colonne C=8 et aura une direction dir égale à GAUCHE.

Une fois lancé, et sans interaction du joueur, Pac-Man se déplace d'une case dans la direction dir, et cela toutes les 0,3 secondes (utilisation de la fonction Attente fournie dans PacMan.cpp) (sans changement de direction pour le moment). Pac-Man est responsable de son propre affichage. Il doit donc mettre à jour le tableau tab, ainsi qu'appeler les fonctions d'affichage nécessaires à la fenêtre graphique. En particulier, la fonction DessinePacMan permet de le dessiner à une case (L,C) bien précise et tenant compte de sa direction de déplacement.

Bien sûr, Pac-Man accède et modifie une variable globale, **tab**, qui sera très bientôt partagée avec d'autres threads… Vous devez donc en protéger l'accès par un mutex : **mutexTab**. Veillez également à bien protéger l'accès à tab lors de l'entrée de Pac-Man et pas uniquement lors de ses déplacements…

Jusqu'à présent, Pac-Man avance dans la direction dir sans aucune interaction du joueur et donc sans changement de direction. Lorsqu'il rencontre un mur, Pac-Man s'arrête mais le thread ne se termine évidemment pas, il attend une modification de la variable dir qui lui permettra de reprendre son parcours dans une autre direction.

Etape 2 : Création du thread Event et déplacement de Pac-Man

Afin de contrôler le thread Pac-Man, le thread principal va lancer le **thread Event** dont le rôle est de gérer les événements provenant du clavier et du clic sur la croix de la fenêtre graphique. Pour cela, le **thread Event** va se mettre en attente d'un événement provenant de la fenêtre graphique. Pour récupérer un de ces événements, le thread Event utilise la fonction **ReadEvent()** de la librairie GrilleSDL. Ces événements sont du type « souris », « clavier » ou « croix de la fenêtre ». Les événements du type « souris » devront être ignorés.

Dans le cas « croix de fenêtre », le **thread Event** doit se terminer par un **pthread_exit()** tandis que le thread principal attend la fin de celui-ci (utilisation de **pthread_join()**) – il n'attend donc plus la fin du thread Pac-Man. Après la fin du thread Event, le thread principal fermera la fenêtre graphique et terminera proprement l'application.

Le **thread Event** attend donc les événements de type « clavier » :

- Dans le cas de la touche ← (KEY_LEFT), il envoie un signal SIGINT au thread Pac-Man.
- Dans le cas de la touche → (KEY_RIGHT), il envoie un signal SIGHUP au thread Pac-Man.
- Dans le cas de la touche ↑ (KEY_UP), il envoie un signal SIGUSR1 au thread Pac-Man.
- Dans le cas de la touche ↓ (KEY_DOWN), il envoie un signal SIGUSR2 au thread Pac-Man.

Vous devez donc armer les signaux SIGINT, SIGHUP, SIGUSR1 et SIGUSR2 afin de modifier la direction de Pac-Man.

Remarquez que, vu que la variable **dir** est globale, elle est accessible dans les handlers de signaux. Pac-Man est donc maintenant capable de se déplacer seul et de changer de direction en suivant les ordres du joueur. Mais il n'a encore rien à manger… Allons-y…

Mais avant cela… Le thread Pac-Man ne peut pas être dérangé (par la réception d'un signal) pendant son attente de 0,3 seconde, et encore moins pendant qu'il met à jour le tableau tab (c'est-à-dire pendant son déplacement proprement dit). Deux **sections critiques** (utilisation de **sigprocmask**) doivent donc

- encadrer l'appel de **Attente()** (afin d'éviter un changement de vitesse de Pac-Man lors d'un changement de direction, cela permet un mouvement plus homogène de Pac-Man), et
- encadrer la mise à jour du tableau tab et la mise à jour de la fenêtre graphique.

Entre ces deux sections critiques, le thread Pac-Man doit vérifier s'il a reçu un signal (de nouveau, utilisation de sigprocmask), afin de vérifier si sa variable dir a été mise à jour.

Etape 3 : Création du thread « Pac-Gom » et mise à jour du thread Pac-Man

Avant de lancer le threadPacMan, le thread principal doit lancer le **threadPacGom**. Celui-ci a pour tâche initiale de remplir le tableau tab de Pac-Gom à toutes les cases libres de la grille de jeu, à l'exception de la case de départ de Pac-Man, c'est-à-dire en (15,8) mais également dans le « futur » nid de fantômes, c'est-à-dire aux cases (8,8) et (9,8). Pour cela, il utilisera la fonction **DessinePacGom**. De plus, il devra placer 4 « super Pac-Goms » aux cases (2,1), (2,15), (15,1) et (15,15) ; pour cela, il utilisera la fonction **DessineSuperPacGom**. Une fois la grille remplie de pac-goms, le threadPacGom doit se mettre en attente de la réalisation de la condition (à l'aide d'un mutex **mutexNbPacGom** et d'une variable de condition **condNbPacGom**) suivante :

« Tant que (nbPacGom) > 0, j'attends… »

où **nbPacGom** est une variable globale représentant le nombre de pac-goms présents dans la grille, que ce soit un pac-gom ou un super pac-gom.

A chaque réveil de sa variable de condition, le **threadPacGom** devra afficher, aux cases (12,22), (12,23) et (12,24) de la fenêtre graphique, le nombre de pac-goms restants dans la grille de jeu. Pour cela, il doit utiliser la fonction **DessineChiffre**.

Une fois qu'il n'y a plus de pac-gom dans la grille de jeu, le **threadPacGom** a trois tâches à réaliser :

1. incrémenter le niveau du jeu de 1 et mettre à jour la fenêtre graphique, c'est donc lui qui est responsable de l'affichage du niveau du jeu à la case (14,22).
2. Le passage d'un niveau à un niveau supérieur doit impliquer que la vitesse de Pac-Man (et celle des fantômes aussi, mais une chose à la fois !) est doublée. Cela se fait par l'intermédiaire d'une variable globale **delai** du type int (nombre de millisecondes). Cette variable doit évidemment être initialisée à 0,3 secondes au début de la partie. A chaque passage vers un niveau supérieur, le threadPacGom doit donc diviser par deux le nombre de nanosecondes

présentes dans la variable delai. Pac-Man doit donc aller lire la variable globale delai avant chacune de ses attentes. Et qui dit variable globale partagée entre threads dit… à vous de réfléchir !

3. Remplir de nouveau la grille de jeu de pac-goms et se remettre en attente sur sa variable de condition.

Retour sur le thread Pac-Man

Maintenant que de la nourriture est disponible, Pac-Man peut manger des pac-goms et des super pac-goms. A chaque fois qu'il avale un pac-gom (ou un super pac-gom), il laisse, après son passage, une case vide. De plus, à chaque fois qu'il mange un pac-gom, le score est incrémenté de 1 ; tandis qu'à chaque fois qu'il mange un super pac-gom, le score est incrémenté de 5. Il modifie donc une variable globale **score** mais ce n'est pas le thread Pac-Man qui est responsable de l'affichage du score, car il ne sera pas le seul à le modifier. Il devra donc réveiller un autre thread, le threadScore (voir étape 4).

A chaque fois qu'il mange un pac-gom ou un super pac-gom, le thread Pac-Man doit décrémenter la variable globale **nbPacGom** et réveiller le threadPacGom (utilisation de **pthread_cond_signal**).

Etape 4 : Création du thread Score

A partir du thread principal, lancer le **threadScore** qui est responsable de l'affichage du score dans la fenêtre graphique. Il doit donc être synchronisé sur toute modification du score. Dès lors, à l'aide d'un couple mutex-variable de condition (**mutexScore**, **condScore**), le **threadScore** se met en attente sur la condition suivante :

« Tant que (MAJScore == false), j'attends… »

En d'autres mots, cela signifie, que tant qu'il n'y a pas de mise à jour de la **variable globale score**, le thread Score attend. **MAJScore** est une **variable globale** booléenne reflétant que le score a été mis à jour par un autre thread. Les 2 variables globales **score** et **MAJScore** sont donc protégées par le même **mutexScore**.

Une fois réveillé, le thread Score doit simplement afficher la valeur de **score** dans les cases (16,22), (16,23), (16,24) et (16,25) de la fenêtre graphique (utilisation de **DessineChiffre()**). Ensuite, il doit remettre **MAJScore** à false avant de remonter dans sa boucle.

Actuellement, seul le thread Pac-Man modifie les variables **score** et **MAJScore** à chaque fois qu'il mange un pac-gom ou un super-pacgom et réveille le threadScore (→ utilisation de **pthread_cond_signal**).

Jusqu'à présent, la vie de Pac-Man est bien heureuse, il mange à volonté, ce qui lui rapporte des points, sa vitesse augmente à chaque changement de niveau, mais il ne meurt jamais donc la partie ne se termine jamais… Mais avant de passer à la création des fantômes…

Etape 5 : Création du thread Bonus

A partir du thread principal, lancer le **threadBonus** dont le rôle est d'insérer aléatoirement un bonus (paire de cerises) sur une case vide de la grille de jeu. Lorsque Pac-Man mange ce bonus, cela lui rapporte 30 nouveaux points au niveau du score.

Pour cela, le **threadBonus** entre dans une boucle dans laquelle il

- se met en attente un laps temps aléatoire compris entre 10 et 20 secondes (utilisation de la fonction **Attente()**),
- recherche une case libre quelconque dans la grille de jeu et y dépose le bonus (macro BONUS) (→ utilisation la fonction **DessineBonus**),
- se met en attente 10 secondes (utilisation de **Attente()**),
- si, au bout des 10 secondes, le bonus est toujours présent, il le retire et remet la case à VIDE ;
- se remet en attente au début de sa boucle.

Etape 6 : Création du thread Compteur de fantômes et des threads fantômes (gestion du mode « fantômes gloutons » ; mode = 1)

A partir du thread principal, lancer le **threadCompteurFantomes** qui est responsable de créer les threads fantômes et de maintenir leur nombre à **8** en permanence (**2 de chaque couleur**). Ce thread doit se mettre en attente (à l'aide d'un mutex **mutexNbFantomes** et d'une variable de condition **condNbFantomes**) sur la condition suivante :

« Tant que les variables nbRouge==2 et nbVert==2 et nbMauve==2 et nbOrange==2, j'attends… »

où **nbRouge**, **nbVert**, **nbMauve** et **nbOrange** sont des variables globales représentant le nombre de fantômes rouges, verts, mauves et oranges respectivement présents dans la grille de jeu à un moment donné. Une fois réveillé, le **threadCompteurFantomes** doit créer un thread fantôme et incrémenter la variable nbXXX correspondante de 1. Venons-en donc aux threads fantômes.

Chaque fantôme est caractérisé par sa couleur, sa position et sa direction courante dans la grille de jeu et donc par la structure suivante :

```
typedef
{
  int L ;
  int C ;
  int couleur ;
  int cache ;
} S_FANTOME ;
```

La direction sera stockée en local dans la fonction du **threadFantome** tandis que le champ cache de la structure sera expliqué plus loin. C'est le rôle du **threadCompteurFantomes** d'**allouer dynamiquement** une structure de ce type pour chaque fantôme créé. Il initialisera la position initiale de chaque fantôme sur leur « nid », c'est-à-dire à la case (9,8). C'est également le **threadCompteurFantome** qui attribue la couleur du fantôme en fonction de ce qui est nécessaire (8 au total donc 2 de chaque couleur). Le champ cache est pour l'instant initialisé à 0. Une fois la structure allouée et initialisée, le **threadCompteurFantome** crée un **threadFantome** en lui **passant en paramètre l'adresse de la structure allouée**.

Une fois démarré, le **threadFantome** placera l'adresse de sa structure (reçue en paramètre) en **zone spécifique** (utilisation de **pthread_setspecific**). Il initialisera sa variable (locale) dir à HAUT avant d'apparaître dans la grille de jeu (en écrivant son propre tid dans la variable tab à sa position → utilisation de **setTab()** de **pthread_self()**). Attention ! On vous demande que les fantômes ne puissent pas se chevaucher ! Veillez à bien protéger l'accès à tab par les fantômes et en particulier lors de leur entrée sur la grille de jeu.

Une fois dans la grille de jeu, les fantômes sont seuls responsables de leurs déplacements. Et ils ne sont pas particulièrement intelligents 😊 ! Ils avancent en ligne droite sans changer de direction jusqu'au moment où il rencontre un mur ou un autre fantôme. Une fois nez à nez avec un mur ou un autre fantôme, un fantôme recherche toutes les directions possibles (lui offrant donc la possibilité de se mettre sur une case différente d'un fantôme ou d'un mur) et pêche au hasard parmi ces possibilités. Entre deux déplacements, un fantôme doit toujours attendre un certain laps de temps correspond à 5/3 du temps d'attente entre deux déplacements de Pac-Man (les fantômes sont donc toujours un peu plus lents que Pac-Man, quel que soit le niveau de jeu). Les threadFantome doivent donc accéder régulièrement (avant chaque attente en fait) à la variable globale **delai** dont on a déjà parlé plus haut… Etant partagée avec Pac-Man et le threadPacGom, son accès doit donc être protégé par… 😊

Dans un premier temps, les fantômes seront dits « gloutons » car ils peuvent manger Pac-Man, et ils seront alors dessinés selon leur couleur et leur direction à l'aide de la fonction **DessineFantome(int l,int c,int couleur,int dir)**. Votre application doit donc comporter une variable globale **mode** égale à 1 pour l'instant, et signifiant que le jeu est en mode « fantômes gloutons ». Le mode « fantômes comestibles » (mode = 2) sera expliqué plus loin. Mais quel que soit le mode du jeu, toute case occupée par un fantôme dans la grille de jeu (tab) sera remplie par la valeur de son **tid**.

Cette dernière remarque fait apparaître un problème… Contrairement à Pac-Man, les fantômes ne mangent ni les pac-goms, ni les super pac-goms, ni les bonus. Donc, les fantômes ne font que les « cacher ». Chaque fantôme doit donc retenir ce qui se trouve en-dessous de lui. Pour cela, il utilise le champ cache de sa variable spécifique qui va mémoriser ce qui se « cache » sous le fantôme, c'est-à-dire soit PACGOM, soit SUPERPACGOM, soit BONUS, soit VIDE. Lors d'un déplacement, le fantôme peut donc « restaurer » la case qu'il libère avec la « valeur » qu'il cachait.

Si la nouvelle case qu'un fantôme doit occuper correspond à PACMAN, il doit « tuer » le **threadPacMan** en utilisant la fonction **pthread_cancel**. Ceci a pour effet de terminer le threadPacMan et par conséquent la partie…

Bon… Le jeu se corse nettement… Pac-Man n'a qu'une seule vie et en plus les fantômes sont gloutons en permanence… Rétablissons donc l'équilibre.

Etape 7 : Donnons 3 vies à Pac-Man 😊 !

A partir de maintenant, ce n'est plus le thread principal qui va lancer le thread Pac-Man. Donc, à partir du thread principal, lancer le **threadVies** dont le rôle est de

- afficher le nombre de vies restant à la case (18,22)
- dans une boucle, lancer et attendre (utilisation de **pthread_join**) 3 threads Pac-Man séquentiellement. A la fin de chaque thread Pac-Man, il remettra à jour le nombre de vies dans la fenêtre graphique.
- Une fois les 3 vies de Pac-Man terminées, il affichera « Game-Over » à la case (9,4) (utilisation de la fonction **DessineGameOver()**) et figera les fantômes qui ne pourront plus bouger. A vous à trouver une solution pour…

Il restera alors au joueur à cliquer sur la croix de la fenêtre (ce qui aura pour effet de terminer le thread Event et donc la fermeture propre de l'application par le thread principal).

Etape 8 : Gestion du mode « Fantômes comestibles » (mode = 2) et création du thread Time Out

Jusqu'à présent, lorsque Pac-Man mangeait un super pac-gom, cela lui donnait 5 points de plus au score. Cela ne change pas mais à partir de maintenant, cela va provoquer le passage du jeu du mode « Fantômes Gloutons » au mode « Fantômes comestibles ». Pour cela, le thread Pac-Man va donc mettre la variable globale **mode** à 2. De plus, il lancera un thread « Time Out » dont le rôle sera de remettre la variable mode à 1 au bout d'un certain laps de temps aléatoire.

Un **threadTimeOut** sera donc lancé par le threadPacMan à chaque fois qu'il mange un super pac-gom. Ce thread générera un nombre aléatoire de secondes compris entre 8 et 15 secondes et exécutera la fonction **alarm** avec ce nombre passé en paramètre, avant de se mettre en attente sur la fonction **pause**. Attention ! Seul le **threadTimeOut** pourra recevoir le signal **SIGALRM**. Dès réception de ce signal, le threadTimeOut remet la variable mode à 1 et se termine.

Oui, mais qu'en est-il si Pac-Man mange un second super pac-gom avant la fin du time out ? Le temps restant de l'ancien time out doit être additionné au nouveau temps aléatoire ! Pour cela, au moment où Pac-Man mange un super pac-gom, il doit vérifier si un **threadTimeOut** est en cours d'exécution. Si c'est le cas, il doit le terminer en lui envoyant un signal **SIGQUIT** mais également annuler l'envoi du signal SIGALRM en utilisant l'appel à **alarm(0)**, qui retourne le nombre de secondes restant… Ce nombre peut alors être passé en paramètre à un nouveau threadTimeOut qui le rajoutera au nombre aléatoire qu'il doit générer.

Jusque maintenant, les fantômes ne changeaient de direction que lorsqu'ils arrivaient nez à nez avec un mur ou un autre fantôme. Il doit maintenant en être de même lorsqu'il tombe nez à nez avec Pac-Man pendant le mode « fantômes comestibles » (mode = 2). De plus, pendant ce mode, les fantômes sont dessinés en bleu (« de peur » 😊). Ceci est simplement réalisé par l'appel de la fonction **DessineFantomeComestible(int l,int c)**.

C'est bien beau, on change maintenant de mode lorsque Pac-Man mange un super pac-gom. Mais que se passe-t-il exactement lorsque Pac-Man mange un fantôme ? Dans le mode « fantôme gloutons » (mode = 1), aucune surprise, le threadPacMan se termine simplement par un pthread_exit. Dans le mode « fantôme comestible » (mode = 2), Pac-Man va gagner des points parce qu'il mange un fantôme mais également les points correspondant à ce qui se trouve sous le fantôme ! Et ça, seul le fantôme le sait ! Donc, pour gérer cela, le **threadPacMan** devra alors envoyer un signal **SIGCHLD** au **threadFantome** correspondant afin qu'il se termine. Pour rappel, le tid du thread fantôme qui vient d'être mangé est disponible dans la variable tab (à la position du fantôme).

Lors de la réception du signal SIGCHLD, le thread fantôme concerné passe par un handler ne contenant que l'appel de **pthread_exit**. Donc, pour la fin d'un **threadFantome**, on vous demande d'écrire une **fonction de terminaison** (utilisation de **pthread_cleanup_push** et **pthread_cleanup_pop**) dans laquelle le **threadFantome**

- incrémente le score de 50, valeur octroyée à Pac-Man pour avoir gobé un fantôme,
- incrémente le score du nombre de points correspondant à ce qui se trouvait sous le fantôme (récupéré via sa variable spécifique) et qui a donc été mangé par Pac-Man en même temps que le fantôme (ne pas oublier de décrémenter nbPacGom dans le cas d'un pac-gom ou super pac-gom !!!),
- réveille le threadScore,
- décrémente la variable **nbRouge** ou **nbMauve** ou **nbVert** ou **nbMauve** de 1 selon sa couleur et réveille le threadCompteurFantomes afin que celui-ci puisse créer un nouveau fantôme.

Dernière chose… En mode fantôme « comestibles », aucun nouveau fantôme ne peut être créé. Une fois réveillé, le **threadCompteurFantomes** doit donc se mettre en attente, à l'aide d'un couple **« mutex-variable de condition »**, sur l'événement

« Tant que mode == 2, j'attends… »

avant de pouvoir recréer des nouveaux fantômes.

Armement et masquage des signaux

L'application fait un usage abondant des signaux. Il est nécessaire de se rappeler que

- l'armement (utilisation de **sigaction**) est propre au processus. Dès qu'un thread a armé un signal, il est armé pour tous les threads et sur le même handler.
- le masquage (utilisation de **sigprocmask**) est propre à chaque thread. Chaque thread doit donc mettre en place son propre masque de signaux afin de ne réagir qu'à ceux qui l'intéressent.

Pour émettre un signal, vous avez le choix entre **kill()** et **pthread_kill()**. Lorsque ce n'est pas précisé dans l'énoncé, vous avez le choix mais attention aux conséquences.

Remarques

N'oubliez pas qu'une variable globale utilisée par plusieurs threads doit être protégée par un mutex. Il se peut donc que vous deviez ajouter un ou des mutex non précisé(s) dans l'énoncé !

Consignes

Ce projet doit être **réalisé sur la machine virtuelle fournie** et par **groupe de 2 étudiants**. Il devra être terminé pour **la date qui sera fixée en temps utile**.

Vous défendrez votre projet oralement et serez évalués **par un des professeurs responsables**.

Et pour ne pas changer les bonnes habitudes :

« **ChatGPT** (ou toute aide « IA » à la production automatique de code) est ton ami mais… Pour qu'il le reste, tu devras être capable d'expliquer tout le code généré par **ChatGPT**. Tout code généré par **ChatGPT** que tu seras incapable d'expliquer en détails débouchera sur une cote nulle concernant la question sur ce code (**ET** pour la partie concernée du projet), même si celui-ci est correct et fonctionnel. »

Bon travail 😊 !