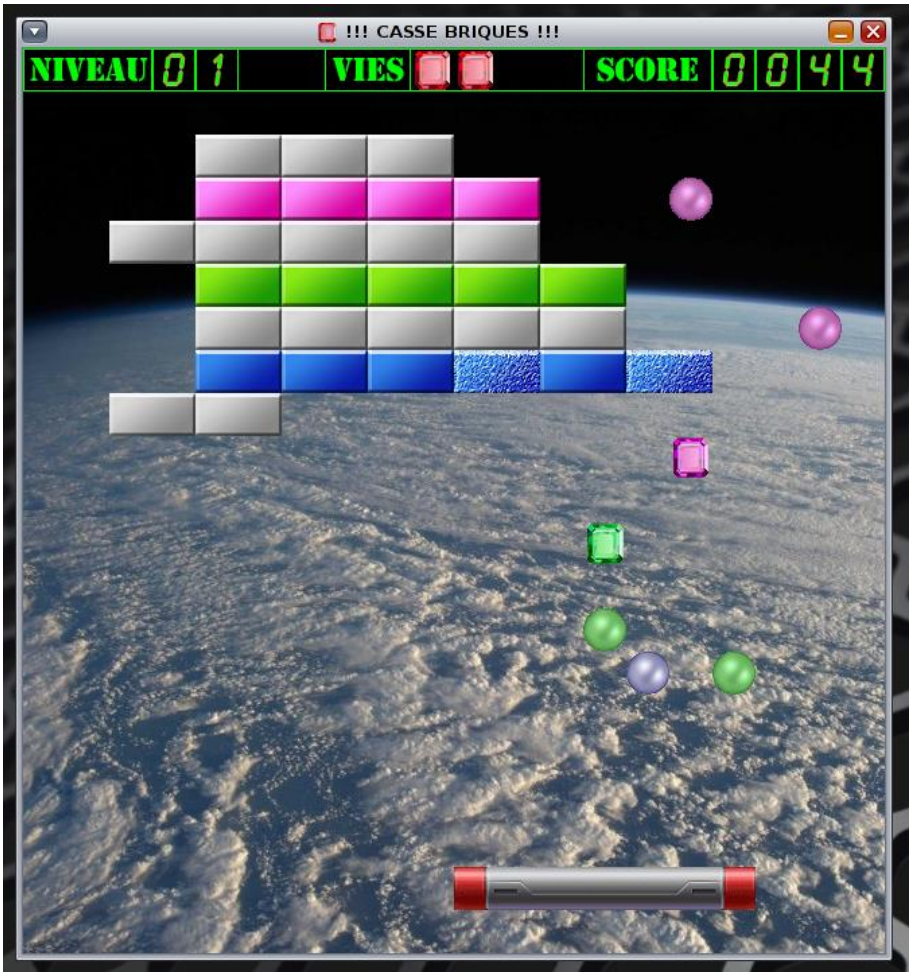
Laboratoire de Threads - Enoncé du dossier final
Année académique 2018-2019
Jeu de « Casse-Briques »

Il s'agit de créer un jeu du type « Casse-Briques » pour un joueur. Le but du jeu est de détruire l'ensemble des briques présentes dans la zone de jeu à l'aide de billes qui se déplacent en oblique. A chaque collision avec une brique grise, une bille fait disparaître celle-ci. Par contre les briques de couleur sont plus résistantes, il faut les toucher deux fois pour qu'elles disparaissent. Les billes rebondissent sur les parois latérales (gauche et droite) et supérieure de la zone de jeu (barre des scores). Par contre, si une bille atteint la paroi inférieure, la bille est perdue. Voilà pourquoi le joueur dispose d'une raquette qu'il peut faire bouger horizontalement afin d'empêcher les billes de sortir de la zone de jeu. Certaines briques cachent des bonus (des diamants colorés) qui apparaissent et tombent verticalement lorsque ces briques sont détruites. Si le joueur attrape le bonus avec sa raquette avant que celui-ci ne quitte la zone de jeu, des effets additionnels se produisent : allongement de la raquette, rétrécissement de la raquette, une bille présente dans la zone de jeu crée deux autres billes qui à leur tour peuvent participer à la destruction des briques. Le joueur dispose de 3 vies. Dès qu'il n'y a plus aucune bille dans la zone de jeu, le joueur en perd une. Si le joueur parvient à casser toutes les briques, il passe au niveau suivant. La différence entre deux niveaux successifs est simplement que les billes se déplacent plus rapidement et donc que le jeu est plus compliqué. A chaque bille détruite et à chaque bonus attrapé, le score est incrémenté de 1. Le but du jeu est donc au final d'accumuler le plus de point possible.

Voici un exemple d'exécution :



Notez dès à présent que plusieurs librairies vous sont fournies dans le répertoire **/export/home/public/wagner/EnonceThread2019**. Il s'agit de

- **Ecran** : la librairie d'entrées/sorties fournies par Mr. Mercenier, et que vous avez déjà utilisée.
- **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, billes, briques, raquettes, diamants, …
- **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 (bille, brique, …) dans la fenêtre graphique.

De plus, vous trouverez le fichier CasseBriques.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. Ce tableau contient des entiers dont le code (macros déjà définies dans CasseBriques.cpp) correspond à

- 0 : VIDE
- -1 : BARRE → zone réservée à la barre des scores/vies/niveau
- -2 : BILLE
- -3 : BONUS
- Une valeur entière positive → tid des briques ou de la raquette

Le tableau tab et la librairie graphique fournie sont totalement indépendants. Dès lors, si vous voulez, par exemple, placer une bille rouge à la case (7,3), c'est-à-dire à la ligne 7 et la colonne 3, vous devrez coder :

```
tab[7][3] = BILLE ;          // gère la logique de tout le jeu
DessineBille(7,3,ROUGE) ;    // fonction du module Ressources,
                             // pour dessiner dans la fenêtre graphique
```

tandis que pour effacer cette bille, il faut coder

```
tab[7][3] = VIDE ;      // gère la logique de tout le jeu
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 Raquette

Le thread Raquette est responsable de l'affichage et du déplacement de la raquette. Les caractéristiques de la raquette sont sa position, sa longueur mais aussi le fait qu'une bille est actuellement collée dessus (cette bille est prête à être lancée). Elle sera donc représentée par la structure

```
typedef struct
{
      int L ;
      int C ;
      int longueur ;
      bool billeSurRaquette ;
} S_RAQUETTE ;
```

où **(L,C)** sont les coordonnées du centre de la raquette et **billeSurRaquette** étant un booléen représentant le fait qu'une bille est collée sur la raquette ou non.

A partir du thread principal, lancez le thread Raquette qui, une fois démarré,

- alloue dynamiquement une structure S_RAQUETTE et initialise ses paramètres avec (L,C) = (19,10), longueur = 5 et billeSurRaquette = false.
- crée une **variable spécifique**, dont la clé est **cleRaquette**, et met en zone spécifique le pointeur correspondant à la structure allouée (utilisation de **pthread_setspecific()**) → n'oubliez pas de créer une fonction « destructeur » de cette variable spécifique !
- met à jour le tableau tab (aux cases occupées par la raquette) avec la valeur du tid (« thread id ») du thread Raquette (utilisation de **pthread_self()**) et se dessine dans la fenêtre graphique à l'aide de la fonction DessineRaquette().
- entre dans une boucle « infinie » dans laquelle il attend un signal à l'aide de l'appel système **pause()**.

Actuellement, la raquette apparaît mais rien ne se passe. Le thread Raquette est en effet en attente d'un signal provenant du thread Event.

Etape 2 : Création du thread Event et déplacement de la raquette

Afin de contrôler le thread Raquette, 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()**). 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 SIGUSR1 au thread Raquette.
- Dans le cas de la touche → (KEY_RIGHT), il envoie un signal SIGUSR2 au thread Raquette.

- Dans le cas de la touche « espace », il envoie un signal SIGHUP au thread Raquette (celui-ci provoquera le lancement de la bille par la raquette → voir étape 3).

Vous devez donc armer les signaux SIGUSR1 et SIGUSR2 afin que la raquette puisse se déplacer horizontalement. Pour cela, les handlers de signaux correspondants

- doivent récupérer la variable spécifique de la raquette (utilisation de **pthread_getspecific()**).
- effacent la raquette et mettent à jour le tableau tab.
- mettent à jour (si c'est possible !) la position de la raquette et redessinent la raquette à la nouvelle position.

Remarquez que la raquette ne peut pas sortir de la zone de jeu, ni par la gauche, ni par la droite, elle « calle » au parois latérales.

Si le booléen **billeSurRaquette** est égal à true, une bille rouge doit être collée au-dessus de la raquette (c'est-à-dire en (18,10) initialement). Tant que ce booléen reste à true, la bille doit se déplacer en même temps que la raquette (et c'est le thread Raquette qui est responsable du déplacement de cette bille). Le joueur pourra en effet choisir la position d'où il veut lancer la bille.

Bien qu'actuellement la raquette est seule à se déplacer dans la zone de jeu et donc à manipuler la variable globale tab, elle ne sera bientôt plus seule et vous devez protéger l'accès à la variable globale tab par le **mutexTab**.

Actuellement, le signal SIGHUP ne doit avoir aucun effet. Celui-ci sera mis à jour lorsque le thread bille aura été mis au point. Par contre, la raquette peut à présent se déplacer horizontalement lorsque vous utilisez les flèches du clavier.

Etape 3 : Création du thread Bille

Le thread Bille est responsable du déplacement d'une bille dans la zone de jeu. Une bille est caractérisée par sa position, sa couleur et sa direction (actuelle) et est représentée par la structure

```
typedef struct
{
      int L ;
      int C ;
      int dir ;
      int couleur ;
} S_BILLE ;
```

où

- **(L,C)** sont les coordonnées de la position actuelle de la bille.
- **dir** est sa direction actuelle. Celle-ci peut prendre les valeurs NO (« Nord-Ouest »), NE (« Nord-Est »), SO (« Sud-Ouest ») ou SE (« Sud-Est »). Ce sont des macros définies dans le fichier CasseBriques.cpp.
- **couleur** est la couleur de la bille et peut prendre une des valeurs ROUGE, VERT, … (macros définies dans Ressources.h)

Au démarrage d'un thread Bille, celui-ci reçoit en paramètre une structure S_BILLE qui aura été allouée dynamiquement par le thread Raquette (voir plus loin). Ce n'est donc pas le thread Bille qui choisit la position, la couleur et la direction initiale de la bille. Une fois démarré, le thread Bille crée une **variable spécifique**, dont la clé est **cleBille**, et met en zone spécifique le pointeur correspondant à

la structure allouée (utilisation de **pthread_setspecific()**). A nouveau, n'oubliez pas de créer une fonction « destructeur » de cette variable spécifique !

Dans le tableau tab, une bille est représentée par la valeur BILLE (macro définie dans CasseBriques.cpp).

Ensuite, la bille entre une boucle dans laquelle elle se déplace toutes les 200 millisecondes (cette valeur évoluera plus tard en fonction du niveau). A chaque tour de boucle, le thread Bille doit comparer sa position par rapport aux cases avoisinantes et mettre à jour sa position en tenant compte de sa direction. Par exemple, supposons que la bille se trouve à une moment donné à la case (L,C) et qu'elle se déplace dans la direction NE. Dans l'ordre, il faut tester

- Si tab[L-1][C] est différent de VIDE, il y a un obstacle au-dessus, la direction change et devient SE.
- Si C == (NB_COLONNES-1), la bille est contre la paroi latérale droite, la direction change et devient NO.
- Si tab[L][C+1] est différent de VIDE, il y a un obstacle à droite, la direction change et devient NO.
- Si tab[L-1][C+1] est différent de VIDE, il y a un obstacle juste à l'endroit où la bille doit se déplacer, elle rebondit en changeant de direction qui devient SO.
- Sinon, rien n'empêche la bille de se déplacer en (L-1,C+1) qui devient sa nouvelle position. Après déplacement, elle peut enfin se mettre en pause de 200 millisecondes (utilisation de la fonction **Attente()** fournie dans CasseBriques.cpp).

Un raisonnement similaire peut-être fait pour les directions NO, SE et SO. Une petite différence cependant pour les directions SE et SO : Si la bille atteint la ligne du bas (L == NB_LIGNES-1), elle ne rebondit pas sur la paroi inférieure (il n'y en a pas !), elle sort de la grille de jeu et le thread Bille se termine par un **pthread_exit()**. De plus, vu qu'à l'endroit de la raquette, le tableau tab est différent de VIDE (il vaut le tid de la raquette), la bille rebondira sur la raquette.

Retour sur le thread Raquette

Pour lancer une bille, le joueur va devoir presser la barre d'espace. Vous devez modifier le handler du signal SIGHUP de telle sorte que lorsque le thread Raquette reçoit ce signal, il

- récupère sa variable spécifique afin qu'il connaisse la position actuelle de la raquette (pour rappel, le joueur peut choisir l'endroit d'où il veut lancer la bille).
- alloue dynamiquement une structure S_BILLE dont la position est (L-1,C) si (L,C) est la position actuelle du centre de la raquette, la couleur est rouge et la direction dir est choisie aléatoirement entre NO et NE (une chance sur deux).
- lance un thread Bille en lui passant en paramètre la structure allouée.

Il est à présent possible de déplacer la raquette et de lancer une bille qui rebondit sur les parois latérales, la barre des scores et la raquette, et qui sort de la grille de jeu par le bas.

Etape 4 : Création du thread Brique

A chaque brique va être associé un thread. Une brique est caractérisée par la structure suivante :

```
typedef struct
{
      int L ;
      int C ;
      int couleur ;
      int nbTouches ;
      int brise ;
      int bonus ;
} S_BRIQUE ;
```

où

- **(L,C)** est la position de la case de gauche de la brique (une brique comporte toujours deux cases consécutives alignées horizontalement).
- **couleur** est sa couleur qui peut prendre une des valeurs ROUGE, VERT, … (macros définies dans Ressources.h).
- **nbTouches** est le nombre de fois qu'il reste à toucher cette brique pour qu'elle disparaisse. Cette variable devra donc initialisée à 1, 2, … avant le lancement du thread Brique.
- **brise** influence son affichage (« neuve » ou « brisée »). Au départ, brise vaut 0 et la brique s'affiche « normalement » (utilisation de la fonction **DessineBrique** avec la valeur brise = 0). Dès que la brique est touchée une première fois, soit elle disparaît (si nbTouches == 0), soit elle doit apparaître « brisée ». Dans ce cas, la variable **brise** passe à 1 et la brique est affichée en utilisant la fonction DessineBrique avec brise = 1.
- **bonus** contiendra (voir plus loin) la couleur du bonus caché par cette brique. Actuellement, cette variable restera égale à 0.

Il y aura donc autant de threads Brique qu'il y a de briques présentes dans la grille de jeu. Dans CasseBriques.cpp, vous trouverez une macro **NB_BRIQUES** fournissant le nombre de briques au départ de chaque niveau (56), tandis que la variable globale **Briques[]** contient les caractéristiques de base de ces 56 briques.

Pour l'instant, c'est le thread principal qui va lancer tous les threads Brique. Donc, à partir du thread principal, et pour chacune des 56 briques,

- allouez dynamiquement une structure S_BRIQUE et copiez-y les caractéristiques de la brique.
- lancez un thread Brique en lui passant en paramètre la structure allouée.

Une fois démarré, le thread Brique

- reçoit en paramètre l'adresse d'une structure S_BRIQUE.
- crée une **variable spécifique**, dont la clé est **cleBrique**, et met en zone spécifique ce pointeur (utilisation de **pthread_setspecific()**) → n'oubliez pas à nouveau de créer une fonction « destructeur » de cette variable spécifique !
- met à jour le tableau tab (aux cases occupées par la brique) avec la valeur du tid (« thread id ») du thread Brique (utilisation de **pthread_self()**) et se dessine dans la fenêtre graphique à l'aide de la fonction DessineBrique().
- entre dans une boucle « infinie » dans laquelle il attend un signal à l'aide de l'appel système **pause()**.

Chaque thread Brique va en effet recevoir un signal SIGTRAP provenant d'une bille qui vient de la toucher. Vous devez donc armer le signal SIGTRAP sur un handler dans lequel le thread Brique « touché » va

- récupérer sa variable spécifique.
- décrémenter sa variable **nbTouches** de 1.
- Si **nbTouches** est à zéro, le thread Brique efface la brique et met à jour le tableau tab avant de se terminer par un **pthread_exit()**.
- Si **nbTouches** est supérieur ou égal à 1, la brique ne disparaît pas. La variable **brise** de la variable spécifique passe à 1 et le thread Brique doit réafficher la brique « brisée » à l'aide de la fonction DessineBrique().

Retour sur le thread Bille

La logique du thread Bille reste totalement identique mis à part que lorsqu'elle rebondit sur un obstacle, c'est-à-dire lorsque tab[…][…] > 0, elle envoie un signal SIGTRAP au thread correspondant (utilisation de **pthread_kill**), c'est-à-dire pthread_kill(tab[…][…],SIGTRAP).

Remarque importante : Attention que seuls les threads Brique peuvent recevoir le signal SIGTRAP. Il est donc impératif de bien gérer vos masques de signaux dans chaque thread !!!

En gros, à présent, le jeu de Casse-Briques est mis en place. Reste maintenant à coordonner le tout !

Etape 5 : Création du thread Niveau

Le **thread Niveau** va être responsable de gérer

- le passage d'un niveau à un autre lorsque toutes les briques ont été détruites, ainsi que l'affichage du numéro du niveau dans la barre des scores.
- l'apparition d'une nouvelle bille lorsque la (les) bille(s) est (sont) sortie(s) de la zone de jeu. Il gère donc le nombre de vies du joueur.
- la mise en place des briques au début de chaque nouveau niveau. Ce n'est donc plus le thread principal qui lance les threads Brique, mais bien le thread Niveau.

A partir du thread principal, lancez le thread Niveau qui, une fois démarré,

- initialise **nbVies** (variable locale) à 2 et affiche 2 diamants rouges aux cases (0,9) et (0,10), ce qui montre au joueur qu'il dispose de 2 vies.
- initialise **niveau** (variable locale) à 1 et mets à jour l'affichage aux cases (0,3) et (0,4).
- entre dans une boucle infinie dont un tour correspond à un niveau complétement terminé.

Dans cette boucle infinie, le thread Niveau

1. mets en place les 56 briques (il fait exactement ce que faisait le thread principal à l'étape 4).
2. initialise la variable globale **nbBriques** à NB_BRIQUES (56). Cette variable représente le nombre de briques présentes dans la grille de jeu à un moment donné.
3. initialise la variable globale **nbBilles** à 1. Cette variable représente le nombre de billes présentes dans la grille de jeu à un moment donné.
4. demande au thread Raquette de se « ré-initialiser » en lui envoyant le signal SIGINT (voir plus loin).
5. initialise la variable globale booléenne **niveauFini** à false.
6. Tant que cette variable reste à false, il tourne dans une boucle dans laquelle il se met en attente de la réalisation de la condition (à l'aide d'un **mutex mutexBilleBrique** et d'une **variable de condition condBilleBrique**) suivante :

« Tant que (nbBilles > 0) && (nbBriques > 0), j'attends… »

Une fois réveillé, il doit vérifier ce qu'il s'est passé :

- **Si nbBilles == 0**, le niveau n'est pas terminé, le joueur a perdu la (les) bille(s) avec la(es)quelle(s) il jouait. Dans ce cas,
  - **Si nbVies == 0**, la partie est terminée. Le thread Niveau affiche « Game Over » dans la fenêtre graphique (utilisation de la fonction **DessineGameOver(9,6)**) avant de se terminer par un **pthread_exit()**.
  - **Si nbVies > 0**, il décrémente nbVies de 1, mets à jour l'affichage dans la barre des scores, remet la variable nbBilles à 1 et demande au thread Raquette de se ré-initialiser en lui envoyant le signal SIGINT. Le Thread Niveau se remet alors en attente sur la variable de condition **condBilleBrique**.
- **Si nbBriques == 0**, le niveau est terminé. Il met la variable globale **niveauFini** à true. Cette variable informe tous les threads Bille restants que le niveau est terminé (voir plus loin). Le thread Niveau doit attendre qu'il n'y ait plus de billes (utilisation de **pthread_cond_wait**) avant de mettre en place le niveau suivant. Dès qu'il n'y a plus de billes en jeu, il incrémente la variable niveau de 1, mets à jour la barre des scores, réalise une temporisation (attente) de 1 seconde avant de diminuer la variable globale **delaiBille** de 10%. La variable globale **delaiBille** est le temps d'attente entre deux déplacements d'une bille. Au départ, elle vaut 200 millisecondes. Dès lors, à chaque niveau terminé, la vitesse des billes sera d'autant plus augmentée.

7. Le niveau étant terminé, il remonte dans sa boucle principale (point 1).

Retour sur le thread Bille

A chaque tour de boucle, le thread Bille doit vérifier si le niveau en cours n'est pas terminé. Pour cela, il regarde la valeur de la variable **niveauFini**. Si cette variable est à true, il doit se terminer par un **pthread_exit()**, après s'être effacé de la fenêtre graphique et avoir mis à jour le tableau tab.

Dès que le thread Bille se termine, il doit décrémenter de 1 la variable globale **nbBilles** et prévenir le thread Niveau (utilisation de **pthread_cond_signal**).

N'oubliez pas de mettre à jour l'attente entre deux déplacements de la bille. La durée de 200 millisecondes est à présent remplacée par la variable globale **delaiBille**.

Retour sur le thread Raquette

Lorsque le joueur perd une vie ou qu'il réussit à passer un niveau, il faut « ré-initialiser » la raquette en la replaçant au centre et en replaçant une bille rouge dessus. Pour cela, il est averti en recevant le signal SIGINT. Vous devez donc armer le **signal SIGINT** sur un handler dans lequel le thread Raquette

- récupère sa variable spécifique.
- s'efface de la fenêtre graphique et met la variable tab à jour.
- remet sa longueur à 5, sa variable billeSurRaquette à true et sa position à (19,10).
- se redessine et met le tableau tab à jour.

Retour sur le thread Brique

Vous devez également mettre à jour le handler du **signal SIGTRAP** de telle sorte que lorsqu'une brique est détruite, le thread Brique décrémente la variable globale **nbBriques** de 1 avant de réveiller le thread Niveau par un **pthread_cond_signal**.

Etape 6 : Création du thread Score

A chaque fois qu'une bille touchera une brique, le score du joueur sera incrémenté de 1. Le score du joueur est représenté par la variable globale (de type int) **score**.

A partir du thread principal, lancer le **thread Score** dont le rôle est d'afficher le score en haut à droite dans la barre des scores.

Une fois lancé, le thread Score entrera dans une boucle dans laquelle il se mettra tout d'abord en attente (via un **mutexScore** et une variable de condition **condScore**) de la réalisation de l'événement suivant

« 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 (0,16), (0,17), (0,18) et (0,19) de la fenêtre graphique (utilisation de **DessineChiffre()**). Ensuite, il doit remettre **MAJScore** à false avant de remonter dans sa boucle.

La question à présent est de savoir qui va réveiller le thread Score. Il s'agit simplement des threads Brique qui modifieront les variables **score** et **MAJScore** à chaque fois qu'ils entrent dans le handler de SIGTRAP (→ utilisation de **pthread_cond_signal**).

Etape 7 : Création du thread Bonus

Nous allons à présent gérer les bonus, c'est-à-dire les diamants colorés cachés dans les briques et qui tombent dès qu'une brique disparaît. Ces bonus vont pimenter le jeu en

- diminuant la taille de la raquette (diamant vert)
- augmentant la taille de la raquette (diamant jaune)
- faisant apparaître des billes supplémentaires dans la zone de jeu en cours (diamant violet)

à condition que le joueur les attrape avec sa raquette avant qu'ils ne sortent (par le bas) de la grille de jeu. En outre, attraper un bonus lui rapportera un point supplémentaire au score.

Un bonus (diamant coloré) est caractérisé par la structure

```
typedef struct
{
      int L ;
      int C ;
      int couleur ;
} S_BONUS ;
```

où

- **(L,C)** est la position actuelle du bonus.
- **couleur** est sa couleur qui peut prendre les valeurs JAUNE, VERT ou MAUVE. Il n'y aura en effet que 3 types de bonus différents (voir ci-dessus).

Retour sur le thread Niveau

La première chose à faire est de « cacher » les bonus dans les briques. Il faut donc modifier le thread Niveau au moment où il met en place les briques. Il doit s'arranger pour cacher aléatoirement NB_BONUS_JAUNE (macro égale à 5), NB_BONUS_VERT (macro égale à 5) et NB_BONUS_MAUVE (macro égale à 10) parmi les NB_BRIQUES (56) qu'il doit placer. Pour cela, il met dans la variable bonus de la structure S_BRIQUE la valeur de la couleur correspondante, avant de lancer chaque thread Brique.

Retour sur le thread Brique

Ce sont les threads Brique qui vont lancer les threads Bonus, et cela quand une brique est détruite, juste au moment où elle disparaît. Pour cela, il faut modifier le handler de SIGTRAP de telle sorte que le thread Brique (juste avant de se terminer), si sa variable bonus est différente de 0 :

- alloue dynamiquement une structure S_BONUS.
- précise la position initiale du bonus qui se cache juste derrière la brique. La brique occupant deux cases, la position initiale du bonus sera choisie aléatoirement entre ces deux cases. La couleur du bonus sera simplement récupérée dans la variable bonus de la variable spécifique du thread Brique.
- lance un thread Bonus en lui passant en paramètre la structure allouée.

Une fois démarré, le **thread Bonus**

- doit tout d'abord attendre que sa position initiale soit VIDE afin de se positionner dans la grille de jeu, puis mettre à jour la variable tab (en mettant la valeur BONUS à sa place, BONUS étant une macro fournie dans CasseBriques.cpp) et s'afficher dans la fenêtre graphique (utilisation de la fonction DessineDiamant).
- entre dans une boucle dans laquelle il va se déplacer verticalement du haut vers le bas avec un intervalle de temps égal à **delaiBille** (il se déplace donc « à la même vitesse que les billes »), et cela jusqu'à, soit sortir (par le bas) de la grille de jeu, soit atteindre la raquette.

Au cours de son déplacement, le **thread Bonus** peut rencontrer plusieurs situations :

- S'il se trouve sur la ligne NB_LIGNES-1 ou que **niveauFini** est égal à true, il doit se terminer. Il se supprime donc de tab, s'efface de la fenêtre graphique et se termine par un **pthread_exit**.
- S'il y a une bille (tab contient la valeur BILLE) ou un autre bonus (tab contient la valeur BONUS) juste en-dessous de lui, il ne se déplace pas, il remonte dans sa boucle afin d'attendre **delaiBille** millisecondes avant de ré-essayer de se déplacer.
- Si la case en-dessous de lui est VIDE ou contient une brique (valeur positive dans tab représentant le tid d'une brique), le bonus peut se déplacer. En effet, un bonus a la faculté de se déplacer par-dessus une brique. Attention que si avant de se déplacer, le bonus se trouvait déjà sur une brique, il doit demander à la brique de se redessiner. Pour cela il lui envoie le **signal SIGURG** (voir plus bas).
- S'il atteint la raquette (il suffit de comparer la valeur de tab à la valeur du tid du thread Raquette), le bonus est acquis par le joueur. Et dans ce cas,
    - il agit en fonction de la couleur du bonus :
        - S'il est VERT, il envoie le **signal SIGSYS** au thread Raquette (voir plus bas).
        - S'il est JAUNE, il envoie le **signal SIGPIPE** au thread Raquette (voir plus bas).
        - S'il est MAUVE, il envoie le **signal SIGEMT** à une des billes déjà présentes dans la zone de jeu (voir plus bas). Pour atteindre (« aléatoirement ») un des threads Bille existants, vous devez utiliser l'appel système **kill()**.
    - il augmente le score de 1 avant de prévenir le thread Score par un **pthread_cond_signal**.

- il se supprime de tab, s'efface de la fenêtre graphique avant de se terminer par un **pthread_exit**.

Les briques doivent se redessiner après le passage d'un bonus

Il est nécessaire qu'après le passage d'un bonus par-dessus une brique de redessiner celle-ci. Le thread Brique correspondant est averti en recevant le **signal SIGURG** (provenant d'un thread Bonus). Vous devez donc armer le signal SIGURG sur un handler dans lequel le thread Brique

- récupère sa variable spécifique.
- se redessine en utilisant la fonction DessineBrique.

Retour sur le thread Raquette

Le thread Raquette est à présent susceptible de recevoir les **signaux SIGSYS** et **SIGPIPE** provenant d'un thread Bonus. Vous devez donc armer ces deux signaux sur des handlers dans lesquels le thread Raquette

- récupère sa variable spécifique.
- annule l'alarme (utilisation de **alarm(0)**) d'un bonus précédent éventuel.
- modifie sa longueur à **3** (SIGSYS) ou à **7** (SIGPIPE). Lors de l'allongement, veillez à ne pas « écraser » une bille ou un bonus. Si la raquette est trop à droite ou trop à gauche, décalez-la afin qu'elle ne déborde pas de la zone de jeu.
- exécute un **alarm(15)**. En effet, la durée de l'effet de l'allongement/rétrécissement est fixée à 15 secondes. Un **signal SIGALRM** sera donc envoyé au processus 15 secondes plus tard. Vous devez veiller à ce que ce soit bien le thread Raquette qui le reçoive.

Vous devez donc armer le signal SIGALRM sur un handler dans lequel le thread Raquette va reprendre sa longueur initiale de **5**.

Retour sur le thread Bille

Un thread Bille est maintenant susceptible de recevoir le **signal SIGEMT** provenant d'un thread Bonus. Vous devez donc armer le signal SIGEMT sur un handler dans lequel le thread Bille va

- récupérer sa variable spécifique afin de connaître sa position (L,C) et sa direction actuelle.
- allouer dynamiquement deux structures S_BILLE dont les positions initiales seront identiques à la position actuelle (L,C) de la bille qui a reçu le signal. Les directions seront choisies perpendiculaires à la direction de la bille qui a reçu le signal. Par exemple, si la direction de la bille est NE, elle va créer deux nouvelles billes qui démarrent dans les directions NO et SE. Leurs couleurs seront choisies aléatoirement.
- incrémenter la variable globale **nbBilles** de 2.
- lancer deux nouveaux threads Billes en leur passant en paramètre les structures allouées.

Armement et masquage des signaux

L'application fait un usage abondant des signaux. Nous allons ici faire un récapitulatif des différents signaux utilisés. Tout d'abord, 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.

Le tableau ci-dessous reprend les différents signaux utilisés, les threads à qui ils sont destinés et leur action.

Signal	Thread	Action
SIGINT	Raquette	La raquette se ré-initialise en se replaçant au centre avec une bille dessus
SIGUSR1	Raquette	La raquette se déplace vers la gauche
SIGUSR2	Raquette	La raquette se déplace vers la droite
SIGHUP	Raquette	La raquette lance la bille qui est « collée » à elle
SIGSYS	Raquette	La raquette rétrécit
SIGPIPE	Raquette	La raquette s'allonge
SIGALRM	Raquette	La raquette reprend sa longueur initiale
SIGTRAP	Brique	La brique est touchée par une bille
SIGURG	Brique	La brique doit se redessiner suite au passage d'un bonus
SIGEMT	Bille	La bille lance deux nouvelles billes

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 dossier doit être **réalisé sur SUN** et par **groupe de 2 étudiants**. Il devra être terminé pour le dernier jour du « 3ème quart ». Les date et heure précises vous seront fournies ultérieurement.

Votre programme devra obligatoirement être placé dans le répertoire **$(HOME)/Thread2019**, celui-ci sera alors **bloqué (par une procédure automatique) en lecture/écriture à partir de la date et heure qui vous seront fournies !**

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

Bon travail ☺ !