Laboratoire de Threads - Enoncé du dossier final
Année académique 2020-2021
Jeu de « Blockudoku »

Il s'agit de créer un jeu du type « Tetris-Like » mais inspiré d'un jeu de sudoku. Le principe fondamental reste identique à celui d'un Tetris normal. Il s'agit de poser des pièces de différentes formes sur une grille de jeu de sudoku (9x9) afin de former des lignes complètes, mais également dans ce cas-ci des colonnes complètes et des carrés 3x3 complets (encadrés de rouge dans la capture d'écran ci-dessous). Les pièces ne tombent pas du haut de l'écran et ne peuvent pas subir de rotation mais peuvent être placées n'importe où sur la grille de jeu à la manière d'un puzzle, et cela à l'aide de la souris. Dès qu'une colonne, une ligne ou un carré 3x3 est rempli(e), celui(celle)-ci disparaît. La partie se termine quand le joueur est incapable de trouver un endroit libre sur la grille de jeu lui permettant de déposer la pièce suivante. L'interface graphique propose également une zone de texte (en bas) permettant de faire défiler des messages d'informations. De plus, en plus du score qui est affiché en haut à droite, le nombre de combos réalisés (nombre de lignes, colonnes et carrés complétés au cours du jeu) est affiché en-dessous de la pièce à insérer dans la grille.

L'application devra être exécutée à partir d'un terminal console mais est gérée par une fenêtre graphique. Toutes les actions que pourra réaliser le joueur seront gérées par la souris.

Voici une capture d'écran du jeu en cours d'exécution :

Le score actuel est de 71 et le joueur a déjà complété et fait disparaître 2 combos (ligne, colonne ou carré 3x3). La « led » verte indique que le joueur peut cliquer actuellement, mais nous y reviendrons...

Notez dès à présent que plusieurs éléments sont fournis dans le répertoire **/export/home/public/wagner/EnonceThread2021**, dont notamment

- **GrilleSDL** : librairie graphique 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, diamants, briques, lettres, chiffres, …
- **Ressources :** Module permettant de charger les ressources graphiques de l'application, de définir un certain nombre de macros propres à l'application et les fonctions permettant d'afficher les différents sprites dans la fenêtre graphique.

De plus, vous trouverez le fichier Blockudoku.cpp qui contient déjà les bases de votre application (dessin de la grille de jeu) et dans lequel vous verrez des exemples d'utilisation 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 surtout 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, et comportant 12 lignes et 19 colonnes. Seules les 9 premières colonnes et les 9 premières lignes seront utilisées pour la partie jouable. Les autres lignes et colonnes serviront à l'affichage du score, de la pièce suivante, etc… Ce tableau contient des entiers dont le code (les macros sont déjà définies dans Blockudoku.cpp) correspond à

- 0 : VIDE
- 1 : BRIQUE
- 2 : DIAMANT

Chaque pièce de Tetris (L,T,J,…) est composée de 1,2,3 ou 4 briques (cases). Les pièces devront être insérées dans la grille de jeu, brique par brique, une brique à chaque clic gauche de la souris.
Donc, le joueur utilisera la souris pour positionner les pièces dans la grille de jeu :

- Un clic gauche fera apparaître, sur une case vide, un diamant correspondant à la pièce suivante à positionner sur la grille. A chaque clic gauche, une seule brique est insérée dans la grille de jeu. Une fois que le joueur aura inséré le nombre de cases correspondant à la pièce en cours dans la grille de jeu et que celles-ci correspondront à la pièce à insérer, les diamants seront remplacés par des briques bleues précisant que la pièce a été correctement déposée et le joueur pourra alors s'attaquer à la pièce suivante. Sinon, les diamants disparaissent.
- Un clic droit fera disparaître toutes les cases actuellement insérées pour la pièce en cours, c'est-à-dire les cases « diamants » et non les briques bleues qui, elles, sont définitivement mises en place.

L'appui sur la croix de la fenêtre graphique permettra de fermer proprement l'application à n'importe quel moment.

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

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

tandis que pour effacer ce diamant, 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 😉). **Le schéma global de l'application est fourni à la dernière page de cet énoncé.**

Etape 1 : Création du threadDefileMessage et d'un premier mutex

Dans un premier temps, le thread principal va lancer le **threadDefileMessage** dont la tâche est de faire défiler en boucle un message dans la zone réservée (cases bleues claires pour l'instant). Pour cela, vous allez déclarer les **variables globales** suivantes :

```
char* message;      // pointeur vers le message à faire défiler
int tailleMessage;  // longueur du message
int indiceCourant;  // indice du premier caractère à afficher dans la zone graphique
```

et utiliser la fonction DessineLettre(). La zone graphique réservée ne contient que 17 cases (ligne 10, colonnes 1 à 17), il est donc impossible d'y afficher un message plus long, d'où la raison du défilement. La variable **indiceCourant** indique simplement l'indice du premier caractère à afficher. Par exemple, si **message** contient « Bienvenue dans Blockudoku » et que indiceCourant vaut 5, la chaine affichée dans la zone graphique est « enue dans Blockud ». Le **threadDefileMessage** doit donc tourner en boucle et incrémenter indiceCourant toutes les 0,4 secondes sans oublier de le remettre à zéro quand il a atteint la fin du message.

On vous demande également de définir une fonction

```
void setMessage(const char *texte,bool signalOn);
```

qui modifiera la variable message et la réallouera en fonction de texte. Remarquez que n'importe quel autre thread (voir plus loin) pourra exécuter cette fonction. Vous devez donc protéger l'accès aux variables globales message, tailleMessage et indiceCourant par un **mutexMessage**.

Si le booléen signalOn est à true, la fonction demandera au système d'envoyer le signal SIGALRM au processus (utilisation de **alarm()**) 10 secondes après avoir modifié le message. Seul le **threadDefileMessage** devra recevoir ce signal. Vous devrez masquer correctement ce signal dans tous les autres threads. Une fois dans le handler de signal, le **threadDefileMessage** modifiera le message afin d'afficher « Jeu en cours » dans la zone de défilement. N'oubliez pas d'annuler une alarme précédente éventuelle en entrant dans la fonction setMessage().

Après avoir lancé le threadDefileMessage, le thread principal initialisera le message à « Bienvenue dans Blockudoku » (signalOn = true ; ce qui aura pour effet d'afficher « Jeu en cours » 10 secondes après le lancement de l'application).

Etape 2 : Création du threadPiece

Dans un second temps, le thread principal va lancer le **threadPiece** dont le rôle est de générer la pièce suivante à placer dans la grille de jeu, puis ensuite d'attendre que le joueur ait inséré suffisamment de cases avant de vérifier la correspondance entre les cases insérées par le joueur et la pièce à insérer. Avant de donner les détails de ce que fait ce thread, regardons comment représenter une pièce.

Une pièce est composée de plusieurs cases (1,2,3 ou 4) et une case est représentée par la structure

```
typedef struct
{
  int ligne;
  int colonne;
} CASE;
```

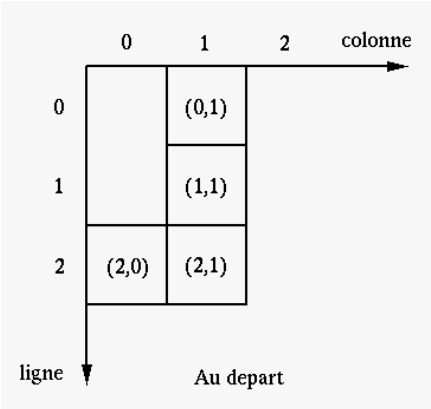
Une pièce est représentée par la structure

```
typedef struct
{
  CASE cases[NB_CASES];
  int  nbCases;
  int  couleur;
} PIECE;
```

où nbCases est le nombre de cases composant chaque pièce (1,2,3 ou 4), et NB_CASES est une macro qui vaut 4. Les 12 pièces possibles vous sont fournies dans le vecteur

```
PIECE pieces[12] = { 0,0,0,1,1,0,1,1,4,0,    // carre 4
                     0,0,1,0,2,0,2,1,4,0,    // L 4
                     0,1,1,1,2,0,2,1,4,0,    // J 4
                     0,0,0,1,1,1,1,2,4,0,    // Z 4
                     0,1,0,2,1,0,1,1,4,0,    // S 4
                     0,0,0,1,0,2,1,1,4,0,    // T 4
                     0,0,0,1,0,2,0,3,4,0,    // I 4
                     0,0,0,1,0,2,0,0,3,0,    // I 3
                     0,1,1,0,1,1,0,0,3,0,    // J 3
                     0,0,1,0,1,1,0,0,3,0,    // L 3
                     0,0,0,1,0,0,0,0,2,0,    // I 2
                     0,0,0,0,0,0,0,0,1,0 } ; // carre 1
```

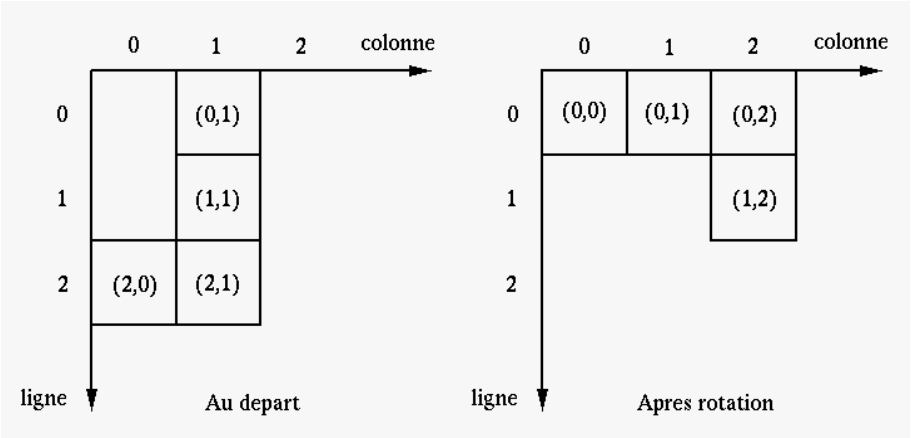
Prenons l'exemple de la pièce « J (à 4 cases) ». Ses cases sont (0,1), (1,1), (2,0), (2,1). Graphiquement, cela correspond à ceci :



Remarquez qu'une pièce ne peut jamais avoir de cases avec des coordonnées (ligne,colonne) négatives et que les cases de la pièce sont triées d'abord sur la ligne puis sur la colonne. Pour simplifier l'algorithme de vérification, il devra toujours en être ainsi. Avant de proposer une nouvelle pièce au joueur, le **threadPiece** fera une ou plusieurs rotations de la pièce de base. Pour cela vous devez utiliser les formules de rotation suivantes d'une case (L,C) autour du point (0,0) :

L (nouveau) = -C (ancienne)
C (nouveau) = L (ancienne)

Comme exemple, reprenons la pièce « J (à 4 cases) ». Après application de ces formules, ses cases sont (-1,0), (-1,1), (0,2), (-1,2). Des coordonnées sont négatives, il va falloir translater la pièce pour les rendre positive. Pour cela, on calcule la ligne minimum et la colonne minimum de chaque case : Lmin = -1 et Cmin = 0. Il suffit donc retirer Lmin et Cmin aux lignes et colonnes de chaque case. Les cases deviennent alors (0,0), (0,1), (1,2), (0,2). Finalement, afin de respecter la contrainte de tri des cases citées plus haut, on trie les cases selon la ligne puis la colonne, ce qui donne : (0,0), (0,1), (0,2), (1,2). Pour cela on vous fournit la fonction **TriCases()** dans le fichier Blockudoku.cpp. Graphiquement, cela correspond à :



Il vous est donc fortement conseillé de concevoir une petite fonction

```
void RotationPiece(PIECE * pPiece);
```

qui réalise cette opération de rotation sur une pièce passée en paramètre.

Nous pouvons à présent revenir sur le détail de ce que fait le **threadPiece**. Une fois démarré, celui-ci

1. choisit une pièce au hasard parmi celles fournies dans le vecteur pieces et la stocke dans une variable globale **pieceEnCours** (qui représente la pièce suivante à insérer). De plus, il lui attribue une couleur (JAUNE, ROUGE, VERT ou VIOLET) au hazard.
2. fait subir à **pieceEnCours** un nombre aléatoire de rotations (0,1,2 ou 3) avant de la dessiner dans la zone graphique qui lui est réservée (lignes 3 à 6, colonnes 15 à 18),
3. se met en attente que le joueur ait inséré suffisamment de cases (voir plus loin),
4. vérifie la correspondance entre les cases insérées par le joueur et la **pieceEnCours** et agira en conséquence (voir étape 3). Selon le cas, il remonte au point 1 (correspondance OK) ou au point 3 (correspondance KO).

Afin de gérer les cases insérées par le joueur, nous allons utiliser les variables globales suivantes :

```
CASE casesInserees[NB_CASES];  // cases insérées par le joueur
int  nbCasesInserees;  // nombre de cases actuellement insérées par le joueur.
```

Donc, pour le point 3, le **threadPiece** va se mettre en attente (utilisation de **pthread_cond_wait**) de la réalisation de la condition (à l'aide d'un mutex **mutexCasesInserees** et d'une variable de condition **condCasesInserees**) suivante :

« Tant que (nbCasesInserees < pieceEnCours.nbCases), j'attends… »

Reste maintenant à permettre au joueur d'insérer des cases… Nous y venons.

Etape 3 : Création du threadEvent

Le thread principal va lancer le **threadEvent** dont le rôle est de gérer les événements provenant de la souris. Pour cela, le threadEvent va se mettre en attente d'un événement provenant de la fenêtre graphique. Pour récupérer un de ces événements, le threadEvent utilise la fonction **ReadEvent()** de la librairie GrilleSDL. Ces événements sont du type « souris », « clavier » ou « croix de la fenêtre ». Les événements du type « clavier » devront être ignorés. Dans le cas « croix de fenêtre », le threadEvent fermera la fenêtre graphique et terminera proprement le processus.

Le **threadEvent** attend donc deux types d'événements provenant de la souris :

- Un clic gauche (event.type == CLIC_GAUCHE) : dans ce cas, le threadEvent récupère la ligne L et la colonne C de la case cliquée dans la grille de jeu. Si tab[L][C] est égal à VIDE, la case est libre, et le threadEvent insère la valeur DIAMANT dans la case correspondante du tableau tab et dessine le diamant correspondant dans la grille de jeu (utilisation de **DessineDiamant**). De plus, il insère la case cliquée dans la variable globale **casesInserees** et incrémente **nbCasesInserees** de 1. Il réveille alors le threadPiece (utilisation de **pthread_cond_signal**). Dans le cas où tab[L][C] n'est pas vide et est donc occupé par une brique bleue ou un diamant, rien ne se passe.

- Un clic droit (event.type == CLIC_DROIT) : dans ce cas, le threadEvent efface les cases dernièrement insérées par le joueur dans la grille de jeu (utilisation de **EffaceCarre()** de GrilleSDL), remet les cases correspondantes de tab à VIDE et remet **nbCasesInserees** à 0.

Retour sur le threadPiece :

Revenons à présent sur le réveil du threadPiece, qui correspond au fait que le nombre de cases insérées par le joueur est égal à pieceEnCours.nbCases. Il peut à présent vérifier la correspondance entre les cases insérées par le joueur et les cases de la pièce en cours. Le plus simple pour cela est de

- trier les cases insérées selon le même critère qu'utilisé pour la pieceEnCours,
- calculer la ligne minimum Lmin et la colonne minimum Cmin de toutes les cases insérées, ce qui permet de translater toutes les cases insérées à « l'origine (0,0) » en soustrayant Lmin des lignes et Cmin des colonnes de toutes les cases insérées.
- La vérification de la correspondance entre les cases insérées et les cases de la pièce en cours devient alors évidente. Par exemple, supposons que le joueur ait cliqué sur les cases (8,4), (8,6), (9,6) et (8,5). Après tri, ces cases deviennent (8,4), (8,5), (8,6) et (9,6). Le calcul des minima fournissent Lmin=8 et Cmin=4. Après soustraction de Lmin et Cmin, nous obtenons (0,0), (0,1), (0,2) et (1,2). Il suffit alors de comparer case par case avec la **pieceEnCours**. On observe ici qu'il y a correspondance entre les cases insérées et la pieceEnCours (à condition que celle-ci corresponde à la pièce « J (à 4 cases) » ayant subi une rotation, voir plus haut).

Après vérification,

- S'il y a correspondance, le threadPiece remplace dans tab les DIAMANT par BRIQUE, signifiant que la pieceEnCours a bien été mise en place. De plus, dans la fenêtre graphique, il remplace les diamants par des cases bleues (utilisation de **DessineBrique()** avec fusion=false). Ensuite, il remet **nbCasesInserees** à zéro et remonte dans sa boucle pour générer une nouvelle pièce à insérer.
- S'il n'y a pas correspondance, le threadPiece remplace dans tab les DIAMANT par VIDE. De plus, dans la fenêtre graphique, il efface les diamants (utilisation de **EffaceCarre()**) et remet **nbCasesInserees** à zéro. Il se remet alors en attente sur la variable de condition **condCasesInserees** (la pieceEnCours n'ayant pas été mise en place, elle reste donc la même).

Jusqu'à présent, le joueur peut positionner les pièces dans la grille de jeu mais les lignes, les colonnes et les carrés complets ne disparaissent pas et il n'y a toujours pas de score. On y vient…

Etape 4 : Création du threadScore

A partir du thread principal, lancer le **threadScore** dont le rôle actuellement est d'afficher le **score** en haut à droite dans la zone graphique correspondante.

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

« Tant que (!MAJScore), 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 threadScore attend. **MAJScore** est une **variable globale** booléenne (bool) 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 threadScore doit simplement afficher la valeur de **score** (variable de type int) dans les cases (1,14), (1,15), (1,16) et (1,17) 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 threadScore. Donc…

Retour sur le threadPiece :

A chaque fois que le threadPiece détecte une correspondance entre les cases insérées par le joueur et la **pieceEnCours**, celle-ci est donc mise en place définitivement dans la grille de jeu. Le threadPiece doit alors incrémenter le **score** du nombre de cases de la pièce mise en place (1,2,3 ou 4) et réveiller le threadScore (**pthread_cond_signal**).

On peut donc à présent, poser les pièces sur la grille de jeu, ainsi que gagner des points au score, mais les lignes, les colonnes et les carrés complets ne disparaissent toujours pas. On y vient…

Etape 5 : Création des threadCases

A partir du thread principal, lancer les **threadCases** dont le rôle est d'analyser/vérifier si la ligne, la colonne et le carré sur lesquels ils se trouvent sont complets, c'est-à-dire remplis de brique. Il faut donc autant de threadCases qu'il y a de cases dans la grille de jeu, c'est-à-dire 9 x 9 = 81. Les identifiants de chaque threadCase seront stockés dans un tableau global de pthread_t :

```
pthread_t tabThreadCase[14][10];
```

Chaque **threadCase** réalisant le même travail mais sur une case qui lui est bien spécifique, chaque threadCase disposera d'une **variable spécifique du type CASE**. Dès lors, dans sa boucle (en fait une double boucle sur L=0…8 et C=0…8) de création des threadCases, le **thread principal**

- alloue dynamiquement une structure CASE qu'il initialisera avec ligne=L et colonne=C,
- crée le threadCase d'identifiant tabThreadCase[L][C] en lui passant en paramètre la structure qu'il vient d'allouer.

Une fois démarré, chaque **threadCase**

- met la variable CASE reçue en paramètre dans sa zone spécifique (utilisation de **pthread_setspecific**),

- entre dans une boucle dans laquelle il attend simplement la réception du signal SIGUSR1 (utilisation de **pause()**) lui demandant d'analyser sa ligne, sa colonne et son carré. Le signal SIGUSR1 lui sera envoyé par le threadPiece (voir plus loin).

Plusieurs threadCases devront faire leurs analyses simultanément. Les résultats de leurs analyses seront stockés dans les **variables globales** suivantes

```
int lignesCompletes[NB_CASES];
int nbLignesCompletes;
int colonnesCompletes[NB_CASES];
int nbColonnesCompletes;
int carresComplets[NB_CASES];
int nbCarresComplets;
int nbAnalyses;
```

L'accès mutuellement exclusif à ces variables globales sera assuré par le **mutexAnalyse**.

A la réception de SIGUSR1, un **threadCase** entre dans un **handlerSIGUSR1** dans lequel :

- il récupère sa variable spécifique (utilisation de **pthread_getspecific**), et donc la ligne L et la colonne C de la case à laquelle il est associé,
- analyse la ligne L dans le tableau tab. Si elle est complète, c'est-à-dire remplie de BRIQUE, il insère L dans le vecteur **lignesCompletes** et incrémente **nbLignesCompletes** de 1, mais cela **à condition que L ne soit pas déjà présent dans le vecteur lignesCompletes**. En effet, un autre threadCase aurait déjà pu traiter cette ligne. En cas de ligne complète, les briques « entrent en fusion ☺ », afin de montrer qu'elles vont bientôt disparaître. Pour cela, dans la fenêtre graphique, le threadCase dessine en fusion chaque brique de la ligne complète (utilisation de **DessineBrique** avec fusion=true).
- analyse la colonne C dans le tableau tab. Si elle est complète, c'est-à-dire remplie de BRIQUE, il insère C dans le vecteur **colonnesCompletes** et incrémente **nbColonnesCompletes** de 1, mais cela **à condition que C ne soit pas déjà présent dans le vecteur colonnesCompletes**. En effet, un autre threadCase aurait déjà pu traiter cette colonne. En cas de colonne complète, les briques « entrent en fusion ☺ », afin de montrer qu'elles vont bientôt disparaître. Pour cela, dans la fenêtre graphique, le threadCase dessine en fusion chaque brique de la colonne complète (utilisation de **DessineBrique** avec fusion=true).
- analyse le carré dans lequel (L,C) se trouve. Il y au total 9 carrés numérotés de 0 à 8 ligne par ligne du haut vers le bas et de la gauche vers la droite. Si le carré correspondant est complet, c'est-à-dire rempli de BRIQUE, il insère le numéro du carré dans le vecteur **carresComplets** et incrémente **nbCarresComplets** de 1, mais cela **à condition que le numéro du carré ne soit pas déjà présent dans le vecteur carresComplets**. En effet, un autre threadCase aurait déjà pu traiter ce carré. En cas de carré complet, les briques « entrent en fusion ☺ », afin de montrer qu'elles vont bientôt disparaître. Pour cela, dans la fenêtre graphique, le threadCase dessine en fusion chaque brique du carré complet (utilisation de **DessineBrique** avec fusion=true).
- incrémente **nbAnalyses** de 1, signifiant qu'il a terminé son analyse. Il réveillera alors (utilisation de **pthread_cond_signal**) le **threadNettoyeur** (voir étape 6).

Vous remarquez donc que les threadCases ne font aucune modification du tableau tab, ce n'est pas leur rôle de supprimer les colonnes, les lignes et les carrés complets (qui sont juste « entrés en fusion ☺ »), ce sera le rôle du threadNettoyeur (voir étape 6).

Retour sur le threadPiece :

Dès que le threadPiece détecte une correspondance entre les cases insérées par le joueur et la pieceEnCours, on sait qu'il incrémente le score du nombre de cases de la pièce insérée. En plus, à partir de maintenant, il enverra le signal SIGUSR1 à chaque threadCase correspondant aux cases de la pièce qui

vient d'être correctement insérée (utilisation de **pthread_kill**, leur pthread_t sont connus dans le tableau **tabThreadCases**), afin de les réveiller pour qu'ils analysent les lignes, les colonnes et les carrés sur lesquels la pièce a été insérée.

Etape 6 : Création du threadNettoyeur

A partir du thread principal, lancer le **threadNettoyeur** dont le rôle est de supprimer les lignes, les colonnes et les carrés en fusion (complets). Dès lors, une fois lancé, il entre dans une boucle dans laquelle il commence par attendre (via le **mutexAnalyse** et une variable de condition **condAnalyse**) sur la réalisation de l'événement suivant

« Tant que (nbAnalyses < pieceEnCours.nbCases), j'attends… »

En d'autres mots, cela signifie, qu'il attend que tous les threadCases concernés (il y en a autant qu'il y a de cases dans la **pieceEnCours**) aient terminé leur analyse.

Une fois son attente terminée, le **threadNettoyeur**

- regarde s'il y a des lignes, des colonnes et/ou des carrés en fusion (complets). S'il n'y a aucune ligne ni aucune colonne ni aucun carré complet, il remet simplement **nbAnalyses** à 0, et remonte dans sa boucle pour se remettre en attente sur sa variable de condition.

- Dans le cas où il y a au moins une ligne et/ou une colonne et/ou un carré en fusion, il commence par attendre 2 secondes (utilisation de **nanosleep**) afin de donner au joueur le temps d'observer les lignes, colonnes et carrés en fusion avant qu'ils ne disparaissent. Il peut ensuite récupérer les résultats d'analyse des threadCases et commencer son travail de suppression. Pour cela, il remettra tab à VIDE aux endroits occupés par les cases en fusion et effacera les cases correspondantes dans la fenêtre graphique (utilisation de EffaceCarre()). Une fois son travail de nettoyage terminé, le threadNettoyeur remet **nbAnalyses** à 0 et se remet en attente sur sa variable de condition.

Enfin, pour chaque ligne/colonne/carré complet supprimé, le **threadNettoyeur** incrémentera une variable globale **combos** de 1 et réveillera le threadScore (voir plus loin). Ce compteur représente le nombre total de lignes/colonnes/carrés complétés et supprimés.

Dans le même temps, pour un cycle de nettoyage, le threadNettoyeur attribuera des points au score :
- 10 points s'il a augmenté la variable **combos** de **1**. De plus il mettra à jour le message à afficher : **« Simple Combo »** pendant 10 secondes (utilisation de setMessage())
- 25 points s'il a augmenté la variable **combos** de **2**. De plus il mettra à jour le message à afficher : **« Double Combo »** pendant 10 secondes (utilisation de setMessage())
- 40 points s'il a augmenté la variable **combos** de **3**. De plus il mettra à jour le message à afficher : **« Triple Combo »** pendant 10 secondes (utilisation de setMessage())
- 55 points s'il a augmenté la variable **combos** de **1**. De plus il mettra à jour le message à afficher : **« Quadruple Combo »** pendant 10 secondes (utilisation de setMessage())

et réveillera le threadScore.

Retour sur le threadScore :

C'est le threadScore qui est chargé de mettre à jour l'affichage du nombre de combos réalisés, et cela aux cases (8,14), (8,15), (8,16) et (8,17) de la fenêtre graphique (utilisation de **DessineChiffre()**). Sa condition d'attente (via le **mutexScore** et la variable de condition **condScore**) devient

« Tant que (!MAJScore) && (!MAJCombos), j'attends… »

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

Une fois réveillé, le threadScore doit simplement afficher la valeur de **score** et/ou **combos** dans la fenêtre graphique avant de remettre le(s) booléen(s) correspondant(s) à false.

Bon… Les lignes, colonnes, carrés complets disparaissent permettant, en plus, d'augmenter le score du joueur, mais que se passe-t-il lorsque le joueur n'a plus de place pour insérer la nouvelle pieceEnCours dans la grille de jeu ? La partie doit se terminer, nous y venons…

Etape 7 : Synchronisation générale du traitement et « led indicatrice »

Vous remarquerez que les threads **threadEvent**, **threadPiece**, **threadCases**, et **threadNettoyeur** travaillent tous à la suite l'un de l'autre (un peu dans le style du modèle pipeline des notes de cours + voir schéma général en fin d'énoncé). Mais le souci ici est que le threadPiece ne peut pas proposer au joueur une nouvelle à insérer tant que toute la chaine de traitement (du threadPiece qui détecte une correspondance jusqu'au threadNettoyeur qui supprime les lignes/colonnes/carrés complets) n'a pas fini son travail.

Pour cela, on vous demande d'utiliser une **variable globale** booléenne **traitementEnCours** protégée par le **mutexTraitement**, et qui sera mise à true pendant tout le traitement (ce qui empêchera le threadPiece de générer une nouvelle pièce et le joueur de cliquer n'importe où) et mise à false lorsque le joueur pourra insérer une nouvelle case.

Graphiquement, tant que le traitement n'est pas en cours (traitementEnCours=false), et donc que le joueur peut insérer une case, la « led » située à la case (8,10) de la fenêtre graphique est **verte** (utilisation de **DessineVoyant()** avec couleur=VERT), signifiant que le joueur peut insérer une nouvelle case. **Tant que le traitement est en cours, la led doit être bleue** (utilisation de **DessineVoyant()** avec couleur=BLEU).

Retour sur le threadPiece :

Après sa vérification de correspondance entre les cases insérées par le joueur et les cases de la pièce en cours, on sait déjà que

- En cas de non-correspondance, le **threadPiece** remet la variable nbCasesInserees à 0 et remonte dans sa boucle se remettre en attente sur la variable de condition **condCasesInserees** (rien à modifier ici).
- A présent, en cas de correspondance, il devra en plus mettre la variable **traitementEnCours** à true et mettre la led en **BLEU**, avant de réveiller les threadCases par l'envoi des SIGUSR1. Ensuite, il doit attendre (via le **mutexTraitement** et une variable de condition **condTraitement**) sur la réalisation de l'événement suivant

  « Tant que (TraitementEnCours), j'attends… »

  Une fois réveillé, il pourra remonter dans sa boucle principale, générer et faire apparaître une nouvelle pièce.

Retour sur le threadEvent :

A partir de maintenant, dès que le joueur cliquera sur une case de la grille de jeu, le threadEvent vérifiera la variable traitementEnCours avant de faire quoi que ce soit.

Si le joueur tente de cliquer
- sur une case déjà occupée (par une brique ou un diamant) que le traitement soit en cours ou pas,
- sur une case vide alors que le traitement est en cours,

la led doit passer au **rouge** (utilisation de DessineVoyant() avec couleur=ROUGE) pendant 0,40 seconde (utilisation de **nanosleep**).

De plus, quoiqu'il arrive, si le joueur clique dans la zone droite de la fenêtre graphique, la led passe au **rouge** pendant 0,40 seconde, avant de reprendre sa couleur précédente (**verte** ou **bleue**).

Retour sur le threadNettoyeur :

Une fois son nettoyage terminé, le **threadNettoyeur** remettra la variable **traitementEnCours** à false, et remettra la led au **vert**. Il pourra ensuite réveiller le threadPiece (utilisation de **pthread_cond_signal**).

Etape 8 : Synchronisation du thread principal et fin de partie

La fin de partie correspond au fait que la nouvelle **pieceEnCours** générée par le threadPiece ne peut trouver une place dans la grille de jeu. Pour cela, on vous demande de modifier le threadPiece de telle sorte, qu'après avoir généré et affiché la nouvelle pièce, il réalise cette vérification : il balaie tout le tableau tab et teste la possibilité d'y insérer la **pieceEnCours**, en vérifiant les cases vides correspondantes. Deux situations peuvent se produire :

1. il trouve une place disponible : il arrête son balayage et continue son exzcution tout à fait normalement.
2. Il ne trouve pas de place disponible : il se termine par un **pthread_exit**, ce qui permettra au thread principal de réagir en conséquence (voir ci-dessous).

Après le lancement de tous les threads, le **thread principal** se synchronisera sur la fin du **threadPiece** (utilisation de **pthread_join**). Une fois celui-ci terminé, le thread principal doit, pour terminer proprement le processus :

- tuer le **threadEvent** (utilisation de **pthread_cancel**) pour empêcher le joueur d'insérer des cases inutilement alors que la partie est terminée,
- tuer l'ensemble des **threadCases** (utilisation de **pthread_cancel**) qui pourront alors passer par leur fonction « destructeur » (fonction mise en place grâce à **pthread_key_create**) qui leur permettra de libérer (**free**) leur variable spécifique. Il attendra qu'ils se soient tous terminés avant de passer au point suivant.
- attendre un événement de type CROIX (utilisation de **ReadEvent**) avant de fermer la fenêtre graphique.
- tuer le **ThreadDefileMessage** (utilisation de **pthread_cancel**). Celui-ci doit d'abord passer par une fonction de terminaison (utilisant de **pthread_cleanup_push** et **pthread_cleanup_pop**) dans laquelle il libère la variable **message** allouée dynamiquement.
- terminer le processus par un exit.

Remarques : signaux et mutex

N'oubliez pas d'**armer** et de **masquer** correctement tous les **signaux** gérés par l'application ! Pour vous aider, voici un tableau récapitulatif des threads de l'application et des signaux qu'ils peuvent recevoir, les autres devant être masqués :

Threads	Signaux pouvant être reçus
Principal	Aucun
threadDefileChaine	**SIGALRM**
threadEvent	Aucun
threadPiece	Aucun
threadCase	**SIGUSR1**
threadNettoyeur	Aucun
threadScore	Aucun

De la même manière, le tableau suivant rappelle les **mutex** principaux et les variables globales qu'ils protègent :

Mutex	Variables globales
mutexTraitement	traitementEnCours
mutexMessage	message, tailleMessage, indiceCourant
mutexCasesInserees	casesInserees, nbCasesInserees
mutexAnalyse	nbAnalyses, lignesCompletes, nbLignesCompletes, colonnesCompletes, nbColonnesCompletes, carresComplet, nbCarresComplets
mutexScore	score, MAJScore, combos, MAJCombos

Mais… 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)/Thread2021**, 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 !

Schéma général de l'application