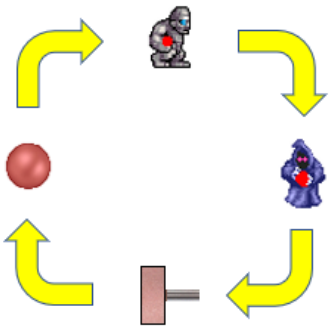
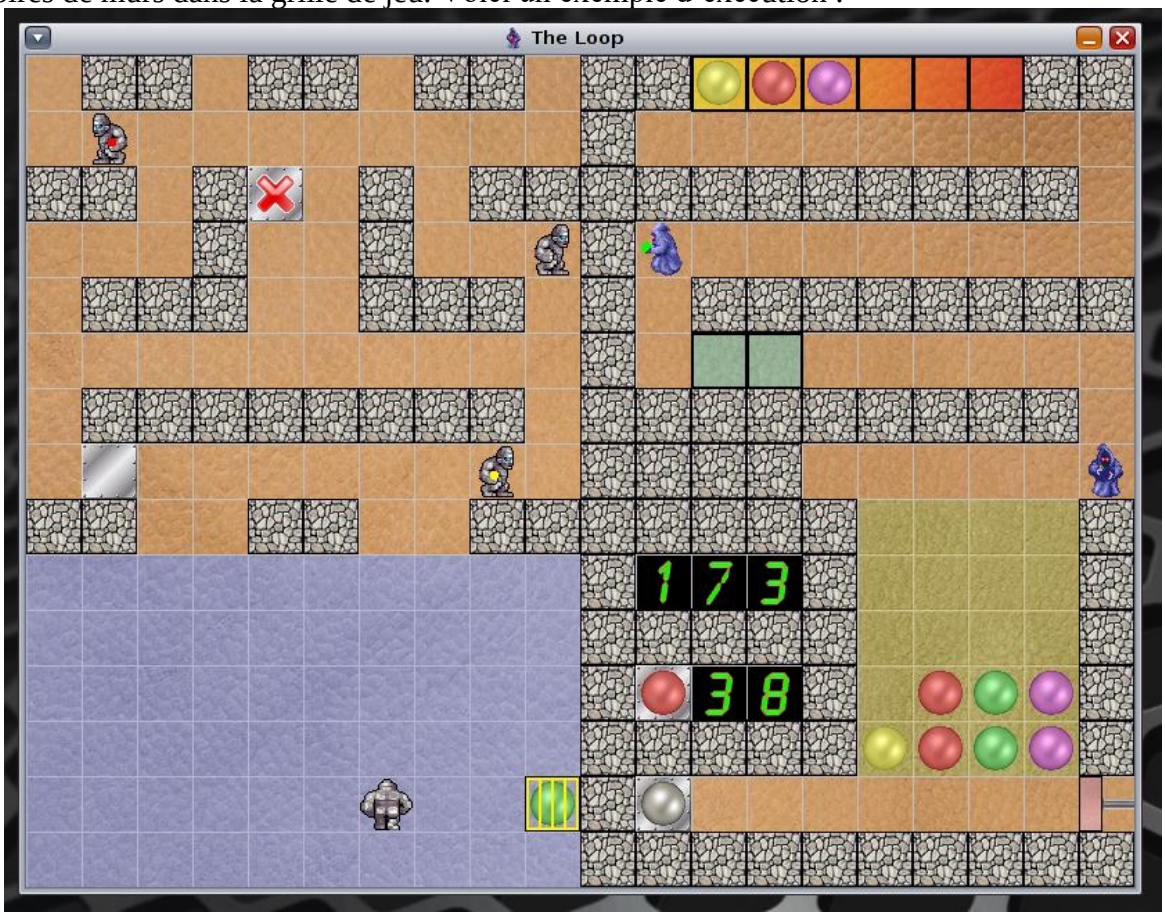
Laboratoire de Threads - Enoncé du dossier final
Année académique 2017-2018
« The Loop »



Il s'agit de créer un mini-jeu dont le contexte est un univers fantastique dans lequel nous verrons apparaître plusieurs intervenants comme des statues mouvantes ou des mages dont la vie est de transporter des billes de couleur. Un ensemble de 60 billes de couleur existe au début du jeu dans un conteneur matérialisé par un compteur à deux chiffres (voir image ci-dessous). Ces billes apparaissent et disparaissent de manière aléatoire dans une zone du jeu (zone bleue en bas à gauche). Le but du joueur est d'attraper chaque bille avant qu'elle ne disparaisse. Une fois attrapée, chaque bille devient la quête d'une statue qui vient la ramasser et la transporter jusqu'à un endroit disponible (zone jaune/orange en haut à droite) par un premier mage. Ce mage la transporte jusqu'à un lieu de rendez-vous (zone verte à droite au milieu) où il la donne à un second mage. Ce dernier la transporte alors jusqu'à une zone où les billes sont triées selon leur couleur et mises en attente dans une zone de transit (zone jaune en bas à droite). Dès que le nombre bille d'une couleur est suffisant (>= 3), un piston se met en mouvement afin de pousser une bille jusqu'au conteneur de départ afin de réalimenter le compteur à deux chiffres. La boucle est bouclée ! Au fur et à mesure que le jeu se déroule, la durée d'apparition des billes dans la zone bleue diminue, ce qui met à l'épreuve vos réflexes et votre rapidité. Le but du jeu est donc de maintenir toute cette belle mécanique en fonctionnement le plus longtemps possible. En effet, si vous ne parvenez pas à attraper une bille, celle-ci disparaît définitivement. Le compteur à trois chiffres indique la durée de votre partie (en secondes) et représente votre score. La partie s'arrête lorsque le compteur de billes tombe à zéro. Enfin, le déroulement du jeu est perturbé par la mise en prison de statues/mages et l'apparition aléatoires de murs dans la grille de jeu. Voici un exemple d'exécution :



Notez dès à présent que plusieurs librairies vous sont fournies dans le répertoire **/export/home/public/wagner/EnonceThread2018**. Il s'agit de

- **Ecran** : la librairie d'entrées/sorties fournie 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, bille, statue, mage, piston, mur, chiffres, …
- **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, statue, …) dans la fenêtre graphique.
- **AStar** : une librairie permettant de calculer le chemin le plus court entre deux cases d'un tableau d'entiers contenant différentes valeurs autorisées ou non. Cette libraire est basée sur l'algorithme de « pathfinding » A*. Elle sera utilisée pour le déplacement des statues, et des mages au sein de la grille de jeu. Des explications sont fournies dans le fichier AStar.h

De plus, vous trouverez le fichier **TheLoop.cpp** qui contient déjà les bases de votre application (ouverture de la fenêtre de jeu) et dans lequel vous verrez des exemples d'utilisation de la librairie GrilleSDL, du module Ressources et de la librairie AStar. 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 est modélisée par un tableau à 2 dimensions (variable **tab**), de taille 15x20, défini en global. Ce tableau contient des entiers dont la valeur correspond à

- 0 : case vide (macro VIDE définie dans TheLoop.c)
- 1 : case occupée par un mur (macro MUR définie dans TheLoop.c)
- (+/-) JAUNE, ROUGE, VERT, VIOLET (macros définies dans Ressources.h) pour une bille non attrapée (+) ou attrapée (-) par le joueur (voir plus bas).
- Une valeur entière positive correspondant au **tid** (« thread id » ou pthread_t) d'un thread Statue/Mage/Mur(destructible) qui occupe cette case (voir plus bas).
- Une valeur entière négative correspondant à **-tid**, où tid est l'identifiant du thread Mur(non destructible) en cette case (voir plus bas).

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

```
tab[7][3] = pthread_self();      // gère la logique de tout le jeu
DessineMage(7,3,DROITE,JAUNE);   // fonction du module Ressources,
                                 // pour dessiner dans la fenêtre graphique
…
tab[7][3] = VIDE ;               // afin de libérer cette case de tab
EffaceCarre(7,3) ;               // fonction de GrilleSDL.h 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**.

Etape 1 : Apparition aléatoire des billes dans la zone bleue (Thread Poseur de Billes et Threads Bille)

Commençons par le début, la génération des billes dans la zone bleue. A partir du thread principal, lancer le **threadPoseurBilles** qui

- Initialise le compteur de billes en mettant la variable globale **nbBilles** à 60. Il met également à jour l'affichage du compteur aux cases (11,12) et (11,13) (utilisation de la fonction **DessineChiffre**).
- Entre dans une boucle dans la laquelle
  - Si **nbBilles** est supérieur à 0, il lance un threadBille en lui passant en paramètre une couleur. Les threads bille sont lancés avec des couleurs définies dans un ordre précis : JAUNE → ROUGE → VERT → VIOLET → JAUNE → … C'est donc le threadPoseurBilles qui choisit la couleur d'une bille mais non sa position.
  - Si **nbBilles** est égal à 0, il s'arrête (utilisation de **pthread_exit**)
  - Décrémente **nbBilles** de 1 et met à jour l'affichage du compteur.
  - Attend (utilisation de **nanosleep**) un nombre aléatoire de secondes compris entre 1 et 10 avant de remonter dans sa boucle.

Une fois lancé, un **threadBille** connait sa couleur et

- Cherche une case libre dans la zone bleue (ligne comprise entre 9 et 14, colonne comprise entre 0 et 9) et y apparaît (utilisation de la fonction **DessineBille**) sans oublier de mettre à jour le tableau tab avec la valeur de la couleur de la bille (macros définies dans Ressources.h)
- Attend un nombre de millisecondes correspondant à la variable globale **attenteBille**. Celle-ci est initialisée au départ à 3000 mais sera mise à jour au cours de la partie.
- Disparaît de la zone bleue (utilisation de **EffaceCarre** située dans GrilleSDL.h) sans oublier de remettre tab à VIDE, et se termine par un **pthread_exit**.

Remarquez que la variable globale **tab** est partagée entre tous les threads et doit donc être protégée (en lecture ET écriture !!!) par un **mutexTab**. De même, la variable globale **nbBilles** sera, à terme, modifiée par plusieurs threads (le threadPoseurBilles mais aussi le threadPiston), elle doit donc être protégée par un **mutexNbBilles**.

Une fois qu'il aura lancé tous les threads (d'autres vont suivre bien sûr…), le thread principal se mettra en attente de la fin du threadPoseurBilles (utilisation de **pthread_join**). La partie étant terminée, le thread principal affichera « GAME OVER » dans la barre de la fenêtre (utilisation de **setTitleGrilleSDL** située dans GrilleSDL.h).

Etape 2 : Gestion des événements de la souris – Nouvelle requête pour les statues (Thread Event)

Afin de permettre au joueur d'attraper les billes en cliquant dessus, le thread principal va lancer le **threadEvent** dont le rôle est de gérer les événements provenant de la souris et du clic sur la croix de la fenêtre graphique. Pour cela, le **threadEvent** va se mettre en attente, dans une boucle infinie 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 » :

- Dans le cas « croix de fenêtre », le **threadEvent** ferme proprement la fenêtre graphique (utilisation de **FermetureFenetreGraphique** de Ressources.h) avant de terminer le processus par un exit.
- Dans le cas où le joueur clique sur un MUR ou dans le VIDE, rien ne se passe.
- Dans le cas où le joueur clique (gauche) sur une bille (ce qu'il détecte si la valeur lue dans tab est JAUNE, ROUGE, VERT ou VIOLET), le **threadEvent**
  - Dessine une prison jaune dessus (utilisation de **DessinPrison** de Ressources.h)
  - Remplace la valeur de tab par l'opposé de la valeur de couleur de la bille attrapée, c'est-à-dire JAUNE devient -JAUNE, ROUGE devient -ROUGE, etc… indiquant que le bille a été attrapée.

Retour sur le threadBille

Une bille qui a été attrapée ne peut plus disparaître. Dès lors, après son attente, le threadBille doit vérifier la valeur de tab à l'endroit où il se trouve :

- Si valeur est positive, la bille n'a pas été attrapée et rien ne change, le threadBille fait disparaître la bille avant de se terminer.
- Si la valeur est négative, la bille a été attrapée et le threadBille doit simplement se terminer sans supprimer la bille de la grille de jeu.

Stockage des billes attrapées : communication entre le threadEvent et les futurs threadStatues

Les billes attrapées vont être stockées dans un conteneur global

```
CASE requetes[NB_MAX_REQUETES + 1] ;
```

où

- CASE est une structure définie dans AStar.h et contenant deux champs entiers L et C, coordonnées d'une case dans la grille de jeu.
- NB_MAX_REQUETES est une macro (égale à 4) qui représente le nombre maximum de requêtes que l'on pourra stocker.

Ce conteneur servira de zone mémoire d'échange de données entre le **threadEvent** et les futurs **threadStatues**. Il doit donc être protégé par un **mutexRequetes**. Ce mutex protègera également les variables globales suivantes :

- **indRequetesE** (type int, au départ égal à 0) qui est l'indice du vecteur **requetes** où le **threadEvent** devra insérer la bille attrapée avant de l'incrémenter de 1. Lorsque indRequetesE == (NB_MAX_REQUETES + 1), celui-ci est remis à 0.
- **indRequetesL** (type int, au départ égal à 0) qui est l'indice du vecteur **requetes** où les futurs **threadStatues** iront lire les coordonnées d'une bille à aller chercher. Après lecture, cet indice sera incrémenté de 1. Lorsque indRequetesL == (NB_MAX_REQUETES + 1), celui-ci est remis à 0. Le conteneur **requetes** est donc un conteneur « circulaire ».
- **nbRequetesNonTraitees** (type int, au départ égal à 0) qui représente le nombre billes qui sont « piégées » dans une prison jaune. Ce nombre ne pourra jamais dépasser NB_MAX_REQUETES.

Retour sur le threadEvent

Lorsque le joueur clique (gauche) sur une bille, le **threadEvent** doit maintenant (en plus de ce qu'il faisait déjà) :

- vérifier la valeur de **nbRequetesNonTraitees** :
  - si **nbRequetesNonTraitees** est inférieure à NB_MAX_REQUETES, il insère la position (structure CASE) de la bille attrapée dans le conteneur **requetes** à l'indice **indRequetesE** avant d'incrémenter cet indice de 1 (voir de le remettre à 0 si nécessaire). La valeur de **nbRequetesNonTraitees** est également incrémentée de 1.

- Si **nbRequetesNonTraitees** est égale à NB_MAX_REQUETES, cette nouvelle requête ne peut être stockée, la bille est perdue et le **threadEvent** dessine une croix rouge dessus (utilisation de **DessineCroix** de Ressources.h) sans modifier la valeur de tab qui reste donc positive.

- Devra réveiller (voir étape suivante) un des threadStatues afin qu'il se mette en quête d'une nouvelle bille (utilisation de **pthread_cond_signal** et d'une variable de condition **condRequetes**).

Retour sur le threadPoseurBilles : Actif / Inactif

On se rend bien compte qu'ainsi, le **threadPoseurBilles** va continuer à poser des billes indéfiniment sans que l'on ne puisse en attraper plus que 4 (NB_MAX_REQUETES). On va donc limiter son comportement en le rendant inactif lorsque trop de billes ont été attrapées. Dès lors, au départ, le **threadPoseurBilles** est actif est pose des billes à intervalles de temps aléatoires. Ceci est visible par le joueur en dessinant une bille rouge à côté du compteur de billes, c'est-à-dire à la case (11,11). Ensuite,

- Dès que **nbRequetesNonTraites** est égale à NB_MAX_REQUETES (4), le **threadPoseurBilles** devient inactif. Ceci est visible par le joueur en dessinant une bille grise à la case (11,11).
- Inactif ne veut pas dire que le **threadPoseurBilles** attend un événement. Il continue à tourner dans sa boucle habituelle, en faisant son attente aléatoire de 1 à 10 secondes mais sans lancer de nouveaux threads bille. Il se contente, à chaque tour de boucle de vérifier la valeur de **nbRequetesNonTraitees**.
  - Tant que **nbRequetesNonTraitees** reste strictement supérieure à NB_MAX_REQUETES / 2 (c'est-à-dire 2), il reste inactif et continue à tourner dans sa boucle sans poser de nouvelles billes.
  - Dès que **nbRequetesNonTraitees** redevient inférieure ou égale à NB_MAX_REQUETES / 2, il redevient actif et recommence, à chaque tour de boucle, à lancer de nouveaux threadBilles. Une bille rouge est alors redessinée à la case (11,11)

Etape 3 : Mise en place du « pool de Threads Statue »

Les statues sont au nombre de 4 et ont pour positions de départ les cases (0,0), (0,3), (0,6) et (0,9). Le thread principal va donc lancer 4 **threadStatues** en leur passant en paramètre une structure CASE contenant leur position de départ.

Une fois démarré, le **threadStatue** met à jour le tableau tab avec la valeur de son tid et dessine la statue orientée vers le bas (utilisation de **DessineStatue** avec le paramètre dir=BAS). Il entrera ensuite dans sa boucle infinie et se mettra en attente de la réalisation de la condition (à l'aide du **mutexRequetes** et de la variable de condition **condRequetes**) suivante :

« Tant que (indRequetesL == indRequetesE), j'attends… »

Les 4 **threadStatues** constituent un « pool de threads » en attente de requêtes provenant du threadEvent. Chaque **threadStatue** aura donc pour tâche d'aller chercher une bille dans la zone bleue et de la ramener à son point de départ (voir plus bas). On dit qu'il est **« consommateur »** de cette tâche, tandis que le threadEvent est le **« producteur »** de tâches. Pour récupérer la prochaine tâche, le **threadStatue** utilise la variable **indRequetesL** pour aller lire la position de la prochaine bille dans la variable globale **requetes**. Une fois que cette position est récupérée en local, il incrémente la variable **indRequetesL** de 1 (éventuellement la remet à 0 si nécessaire).

Une fois que la position d'une bille a été lue dans le conteneur **requetes**, le **threadStatue**

- Se déplace jusqu'à la bille à récupérer. Pour cela, il va utiliser la fonction **deplacement** décrite ci-dessous.
- Récupère la couleur de la bille : elle correspond à l'opposé de la valeur de tab à cet endroit et sera obtenue par la valeur de retour de la fonction **deplacement**. Après son départ, le **threadStatue** remettra tab à VIDE à l'endroit où il a récupéré la bille.
- Attend 1600 millisecondes avant de décrémenter la variable **nbRequetesNonTraitees** de 1.
- Se déplace jusqu'à sa position de départ en utilisant la fonction **deplacement** décrite ci-dessous.
- Une fois arrivé à sa position de départ, il transmettra la bille récupérée au premier Mage mais cela sera décrit plus tard.
- Il se remet en attente (d'une nouvelle requête) sur la variable de condition.

La fonction deplacement : mise en place d'une variable spécifique

On se rend bien compte qu'il n'y a pas que les statues qui vont devoir se rendre d'une case à une autre au sein de la grille de jeu. Il en sera de même pour les mages. Il serait donc judicieux de mettre en place une fonction **déplacement** qui sera utilisée simultanément par les threadStatues et les futurs threads Mages. Pour cela, on vous demande de mettre en place une **variable spécifique** pour les statues (également pour les Mages mais on en reparlera plus loin). On définira donc la structure suivante :

```
typedef struct {
      int id ;
      CASE position ;
      int bille ;
} S_IDENTITE ;
```

où

- **id** est l'identité d'un thread, variable qui peut prendre la valeur STATUE ou MAGE (macros définie dans TheLoop.c). Attention id n'est pas la valeur de pthread_self() !!!
- **position** est la position actuelle du personnage dans la grille de jeu.
- **bille** est une variable représentant le fait que le personnage porte actuellement une bille ou non. Elle peut prendre les valeurs JAUNE, ROUGE, VERT ou JAUNE si une bille est actuellement transportée ou 0 sinon.

Une fois démarré, le **threadStatue** allouera dynamiquement une structure S_IDENTITE qu'il mettra en zone spécifique (via une clé unique pour tous les threads → rappel important !) (utilisation de **pthread_setspecific**). Cette structure permettra à un thread de savoir qui il est quelle que soit la fonction où il se trouve, et notamment dans la fonction **deplacement**.

A présent, on vous demande d'écrire la fonction

```
int deplacement(CASE destination,int delai) ;
```

qui réalise les actions suivantes :

- elle récupère la **variable spécifique** du thread (utilisation de **pthread_getspecific**)
- elle entre dans une boucle dans laquelle elle doit, à chaque tour, utiliser la fonction **RechercheChemin** de la librairie AStar qui vous est fournie. Cette fonction retourne une liste de cases à parcourir pour atteindre la destination :
  - Si un chemin est disponible (retour de **RechercheChemin** non NULL), le personnage se déplace d'une case puis attend **delai** millisecondes.
  - Si aucun chemin n'est disponible (retour de **RechercheChemin** NULL), ce qui peut arriver si le chemin est bloqué par un mur ou une statue (les statues ne peuvent pas se

chevaucher !), le personnage doit attendre **delai** millisecondes avant de remonter dans sa boucle et appeler à nouveau **RechercheChemin**.

- Dès que le personnage atteint la **destination**, il récupère la valeur de tab et retourne cette valeur après avoir remis tab à VIDE.

Remarques :

- Au cours de son déplacement vers sa **destination**, une statue/un mage doit tenir compte de sa direction et du fait qu'il porte une bille ou non pour s'afficher (utilisation correcte de **DessineStatue** ou **DessineMage** avec les bons paramètres dir et couleur).
- Au fur et à mesure du déplacement du personnage, la position du personnage (située dans la **variable spécifique**) doit être mise à jour.

Note : Lors de l'appel à la fonction **RechercheChemin**, il est possible de préciser les cases sur lesquelles peut passer un personnage, et cela dans le paramètre **valeursAutorisees** de la fonction. Dans notre cas, la seule valeur autorisée pour les statues et les mages est VIDE.

Echange de bille entre Statues et le premier Mage

Dès qu'une statue est retournée à sa case départ en possession d'une bille, il doit la transmettre au premier Mage. Pour cela, une zone d'échange (la zone jaune/orange) va être mise en place sous la forme d'une **pile de 6 cases**. Les cases de tab correspondant à cette pile sont (0,12), (0,13), (0,14), (0,15), (0,16) et (0,17). Le sommet de cette pile est toujours la case la plus à droite et la pile croit donc vers la droite et décroit vers la gauche.

Cette pile étant partagée entre les 4 **threadStatues** et le futur **threadMage1** (voir plus loin), elle doit être protégée par le **mutexPile**. Dès lors, dès qu'une statue est de retour à sa case départ chargée d'une bille, elle doit

- Insérer la bille sur le sommet de la pile en mettant à jour tab et dessinant la bille dans la case correspondante. A chaque insertion sur la pile, la variable globale **nbPile** est incrémentée de 1.
- Si la pile est pleine, la statue attend une seconde avant de réessayer. On remarque donc que tant que la pile est pleine, il est impossible à une statue d'aller chercher une autre bille !
- Une fois la bille déposée sur la pile, la statue doit réveiller la threadMage1 par l'intermédiaire de la variable de condition **condPile** (utilisation de **pthread_cond_signal**) et peut à présent se remettre en attente sur sa variable de condition ou aller chercher une nouvelle bille attrapée.

Etape 4 : Création des Mages (Threads Mage 1 et 2)

Le rôle du premier Mage est de ramasser une bille au sommet de la pile (zone jaune/orange) et de la transporter jusqu'à un lieu d'échange (zone verte de 2 cases à droite) où il la donnera au second Mage. Le second mage transporte la bille jusqu'à une zone de tri des billes selon leur couleur, modélisée par 4 files (disposées verticalement dans la grille de jeu).

A partir du thread principal, lancer le **threadMage1** qui

- Se positionne sur sa case départ (1,11) et se dessine orienté vers la droite et sans bille.
- Initialise sa **variable spécifique**, de type S_IDENTITE, comme l'ont fait les threads Statues.
- Entre ensuite dans une boucle infinie et se met en attente de la réalisation de la condition (à l'aide du **mutexPile** et de la variable de condition **condPile**) suivante :

« Tant que (nbPile == 0), j'attends… »

- Une fois réveillé, le **threadMage1** se déplace jusqu'à la case située juste en-dessous du sommet de la pile (sans empêcher les statues d'ajouter d'autres billes à la pile), récupère la bille et décrémente **nbPile** de 1.
- Il prévient le **threadMage2** en lui envoyant le signal SIGHUP (utilisation de **pthread_kill**).
- Il se déplace (utilisation de **deplacement**) jusqu'à la case (5,12) où aura lieu l'échange de bille avec le **threadMage2** (voir plus bas).
- Il retourne à sa case départ (1,11) et se redessine orienté vers la droite et sans bille.
- Il remonte dans sa boucle et se remet en attente sur sa variable de condition ou se met en route vers le sommet de la pile.

A partir du thread principal, lancer le **threadMage2** qui

- Se positionne sur sa case départ (7,14) et se dessine orienté vers la droite et sans bille.
- Initialise sa **variable spécifique**, de type S_IDENTITE, comme l'ont fait les threads Statues.
- Entre ensuite dans une boucle infinie et attend le signal SIGHUP (utilisation de **pause**, **sigprocmask** et **sigaction,** ou alors beaucoup plus simple… **sigwait**).
- Une fois réveillé, il se déplace (utilisation de **deplacement**) jusqu'à la case (5,13) où aura lieu l'échange bille avec le **threadMage1** (voir plus bas).
- Il se déplace jusque l'entrée de la bonne file, (7,15) pour une bille jaune, (7,16) pour une bille rouge, (7,17) pour une bille verte et (7,18) pour une bille violette.
- Il « lance » la bille à l'entrée de la file correspondante (voir plus loin), retourne à sa case départ et se redessine orienté vers la droite et sans bille.
- Il remonte dans sa boucle et se remet en attente d'un signal SIGHUP provenant du threadMage1 ou, s'il l'a déjà reçu, se met en route vers la zone d'échange.

Quelque soit le mage, un délai de 200 millisecondes est imposé entre chaque case lors du déplacement.

Echange de bille entre le Mage 1 et le Mage 2

L'échange va se réaliser à l'aide d'une variable globale **billeEchange** dont l'accès sera synchronisé par deux mutex : **mutexEchangeL** et **mutexEchangeE**. Au démarrage de l'application, le thread principal doit

- Locker le **mutexEchangeE** (utilisation de **pthread_mutex_lock**) : le **threadMage1** sera bloqué dessus tant qu'il ne pourra pas déposer sa bille dans **billeEchange**.
- Locker le **mutexEchangeL** (utilisation de **pthread_mutex_lock**) : le **threadMage2** sera bloqué dessus tant qu'il n'y a rien dans **billeEchange**.

Lors de l'échange, les deux mages se regardent face à face et la séquence des événements est la suivante :

1. Le **threadMage1** arrive à la zone d'échange et tente de locker **mutexEchangeE**.
2. Le **threadMage2** arrive à la zone d'échange et libère **mutexEchangeE** (utilisation de **pthread_mutex_unlock**) et tente de locker **mutexEchangeL**.
3. Pouvant à présent locker **mutexEchangeE**, le **threadMage1** dépose sa bille dans **billeEchange**, se redessine sans bille et libère **mutexEchangeL**.
4. Pouvant à présent locker le **mutexEchangeL**, le **threadMage2** récupère la bille hors de **billeEchange** et se redessine avec la bille.

L'échange est à présent terminé et les deux mutex ont retrouvé leur état initial, prêts pour un autre échange. Remarquez que les étapes 1 et 2 peuvent être inversées selon l'ordre d'arrivée des mages à la zone d'échange mais cela n'a aucune importance, la synchronisation est assurée.

Lancement d'une bille dans la bonne file

Il y a donc 4 files de 5 places, une par couleur (C=15 pour jaune, C=16 pour rouge, C=17 pour vert et C=18 pour violet). Quelle que soit la couleur, l'entrée d'une bille dans la file se fait à la ligne 8 et la tête de la file est à la ligne 12. Les nombres de billes dans chaque file sont représentées par les variables globales **nbFileJaune**, **nbFileRouge**, **nbFileVert** et **nbFileViolet**. Ces 4 variables globales et la zone de tab correspondant aux files (L=8 à 12 et C=15 à 18) seront protégées par le même **mutexFile**.

Une fois arrivé devant l'entrée de la bonne file, le **threadMage2** se dessine face vers le bas et doit attendre (comme vous voulez…) que le nombre de billes dans la file soit inférieur à 5 (sinon la file est pleine et il doit attendre que le futur piston fasse son œuvre, voir plus loin). Dès que c'est possible, il lâche la bille dans la file. Pour cela, il lance un **threadBilleQuiRoule**, en lui passant en paramètre sa couleur, dont le rôle est de

- Incrémenter le bon compteur de file (**nbFileJaune**, **nbFileRouge**, **nbFileVert** ou **nbFileViolet**).
- Se déplacer verticalement (vers le bas) dans sa file d'attente afin d'aller se mettre à la bonne position (soit à la ligne 12 en tête de file, soit juste derrière la bille précédente de même couleur).
- Réveiller le futur threadPiston par l'intermédiaire de la variable de condition **condFile** (utilisation de **pthread_cond_signal**, ou **pthread_cond_broadcast** selon votre cas).

Pendant ce temps, le **threadMage2** est retourné à sa case départ avant de remonter dans sa boucle.

Etape 5 : Création du Thread Piston

Le rôle du piston est d'extraire une bille d'une des files suffisamment remplie et de la pousser jusqu'à bout de course afin d'alimenter le conteneur initial de billes et d'incrémenter le compteur correspondant. A partir du thread principal, lancer le **threadPiston** qui, après avoir positionné sa tête (utilisation de **DessinePiston**) en (13,19), entre dans une boucle dans laquelle

- Il se met en attente de la réalisation de la condition (à l'aide du **mutexFile** et de la variable de condition **condFile**) suivante :

« Tant que (nbFileJaune < 4) et (nbFileRouge < 4) et (nbFileVert < 4) et (nbFileViolet < 4), j'attends… »

On remarque donc que, tant qu'une des files ne contient pas au moins 4 billes, le piston ne fait rien.

- Une fois réveillé, il se déplace jusqu'à la file correspondante (avec un délai de 600 millisecondes entre chaque déplacement). Au fur et à mesure que la tête du piston se déplace vers la gauche, on voit apparaître la tige du piston (utilisation de **DessinePiston**). N'oubliez pas de mettre à jour la valeur de tab avec la macro PISTON (définie dans TheLoop.c)
- Une fois arrivé face à la tête de la file correspondante, il fait descendre la première bille de la file, décale les autres vers le bas et décrémente de 1 la valeur **nbFileXXX** correspondante.
- Il pousse ensuite la bille jusqu'à la case (13,11) où la bille grise prend alors la couleur de la bille arrivée pendant 700 millisecondes avant de redevenir grise. Le compteur **nbBilles** est alors incrémenté de 1 et l'affichage du compteur est alors mis à jour.
- Sa tâche étant terminée, le piston se déplace vers la droite, jusque sa case départ (13,19) avant de remonter dans sa boucle.

Etape 6 : Un peu d'imprévu 😊 : Mise en prison des Statues et des Mages
Gestion de SIGUSR1 et SIGALRM – Section critique

Afin de pimenter un peu le jeu, les statues et les mages vont être mis en prison :

- Soit par l'intermédiaire du joueur qui aura malencontreusement cliqué dessus (gestion du signal SIGUSR1) → prison jaune
- Soit de manière aléatoire suite à la réception du signal SIGARLM → prison verte

Gestion de SIGUSR1

Au fur et à mesure de l'évolution du jeu, le joueur va devoir faire preuve de plus en plus d'adresse et cliquer de plus en plus vite afin d'attraper les billes. S'il clique par erreur sur une statue ou un mage, celui (celle)-ci va se retrouver emprisonné(e) dans une prison jaune (un peu comme les billes) à l'endroit où il se trouve pendant un certain temps.

Pour cela, nous devons revenir sur le **ThreadEvent**. Au moment du clic, s'il détecte que la valeur de **tab** est positive et différente de JAUNE, ROUGE, VERT et VIOLET, il s'agit du **tid** d'un **threadStatue** ou d'un **ThreadMage** (et bientôt d'un **threadMur**, voir plus loin). Dès lors, il lui envoie le signal SIGUSR1 (utilisation de **pthread_kill**).

Donc, vous devez armer correctement le signal SIGUSR1 (utilisation de **sigaction**) afin que le **threadStatue** ou le **threadMage** correspondant entre dans le HandlerSIGUSR1 dans lequel

- Il récupère sa variable spécifique de type S_IDENTITE.
- Il se dessine (utilisation de **DessineStatue** ou **DessineMage** selon le cas) face vers le bas et dessine une prison jaune par-dessus (utilisation de **DessinePrison**).
- Il attend 10 secondes.
- Il se redessine face vers le bas mais sans la prison avant de reprendre sa route.

Attention !!! Les threads Mages et les threads Statue ne peuvent pas être interrompus par SIGUSR1 à n'importe quel moment !!! Vous devez donc mettre en place une **section critique** (utilisation de **sigprocmask**) afin que ces threads ne soient interrompus par SIGUSR1 que lorsqu'ils se trouvent dans la fonction **deplacement** et en-dehors de la section critique mise en place par le **mutexTab**.

Gestion de SIGALRM

Le principe est ici pratiquement identique au SIGUSR1, à la différence près que le signal va être envoyé par le système. Dès lors, lorsqu'il aura lancé tous ses threads, le principal exécutera la fonction **alarm(5)**, ce qui aura pour effet d'envoyer, 5 secondes plus tard, un SIGALRM au processus.

Vous devez donc armer correctement le signal SIGALRM (utilisation de **sigaction**) et masquer correctement ce signal (utilisation de **sigprocmask**) afin que seul un **threadStatue** ou un **threadMage** puisse le recevoir et entrer dans le HandlerSIGALRM dans lequel

- Il récupère sa variable spécifique de type S_IDENTITE.
- Il se dessine (utilisation de **DessineStatue** ou **DessineMage** selon le cas) face vers le bas et dessine une prison verte par-dessus (utilisation de **DessinePrison**).
- Il attend 5 secondes.
- Il se redessine face vers le bas mais sans la prison.
- Il exécute un nouvel appel à la fonction **alarm** avec un nombre aléatoire de secondes compris entre 5 et 15, avant de reprendre sa route.

Attention !!! A nouveau, les threads Mages et les threads Statue ne peuvent pas être interrompus par SIGALRM à n'importe quel moment !!! Vous devez donc mettre à jour la **section critique** mise en place pour SIGUSR1 afin qu'il en soit de même pour SIGALRM et SIGUSR1.

Remarquez que les handlers de SIGUSR1 et SIGALRM sont très similaires, libre à vous d'en faire un seul pour l'armement des deux signaux en les différenciant grâce au paramètre sig de la fonction handler.

Etape 7 : Un peu plus d'imprévu 😤 : Apparition aléatoire de murs (Thread Poseur de Mur et Threads Mur)

Toujours pour pimenter un peu plus le jeu, des murs métalliques vont apparaître de manière aléatoire dans les zones de déplacement des statues, des mages mais aussi du piston qui voit ainsi sa course bloquée. Une fois un certain temps passé, une petite croix rouge apparaîtra par-dessus, ce qui indiquera au joueur qu'il pourra cliquer dessus afin de le faire disparaître.

A partir du thread principal, lancer le **threadPoseurMur** qui entre dans une boucle dans laquelle

- Il attend un temps aléatoire compris entre 10 et 20 secondes.
- Lance un thread Mur avant de remonter dans sa boucle.

Une fois démarré, le **threadMur**

- Cherche une position aléatoire VIDE dans la grille de jeu. Pour cela, il aura 2 chances sur 3 de se positionner dans la partie gauche de la grille de jeu (C compris entre 0 et 9, zone des statues) et 1 chance sur 3 de se retrouver dans la partie droite de la grille de jeu (C compris entre 11 et 19, zone des mages et du piston). De plus, il ne pourra pas se positionner n'importe où, comme par exemple sur les cases départ des statues/mages, la pile, les files ou la zone d'échange. Pour éviter cela, on vous fournit la fonction **CaseReservee** permettant de tester si une case est réservée et ne peut donc pas être un lieu d'apparition d'un mur.
- Se dessine (utilisation de **DessineMur** avec type = METAL) et met à jour **tab** avec l'**opposé de son tid**. Ceci évitera, dans un premier temps, que le **threadEvent** puisse lui envoyer le signal SIGUSR1. Dès lors, pour l'instant, un clic sur ce mur n'aura aucun effet.
- Attend 10 secondes puis met à jour **tab** avec la **valeur de son tid**. Cette valeur étant positive, le **threadEvent** pourra lui envoyer un SIGUSR1 dès que le joueur aura cliqué dessus.
- Attend à présent le signal SIGUSR1 (comme vous voulez, mais **sigwait** est tout de même gentil…).
- S'efface et met à jour **tab** à VIDE avant de se terminer par un **pthread_exit**.

Bien sûr, vous devrez tenir compte de la présence de ces murs dans les déplacements des statues et des mages, mais bonne nouvelle la fonction **RechercheChemin** va faire le travail pour vous. Par contre, vous devez mettre à jour le **threadPiston** pour l'empêcher d'avancer si un mur le bloque.

Etape 8 : Gestion du « Score – Chrono » (Thread Chrono)

Comme on l'a compris, le but du jeu est de maintenir toute cette « belle boucle mécanique » en fonctionnement le plus longtemps possible. Il est donc logique que le score du joueur soit la durée de la partie, entre le début et le moment où il n'y a plus de bille dans le compteur de billes. C'est le Thread Chrono qui sera responsable de gérer le score mais également de mettre à jour, toutes les minutes, le temps d'apparition des billes dans la zone bleue. Il est donc responsable de l'augmentation graduelle de la difficulté du jeu.

A partir du thread principal, lancer le **threadChrono** qui

- Initialise le score à 0 et l'affiche dans les cases (9,11), (9,12) et (9,13) (utilisation de **DessineChiffre**).
- Entre dans une boucle dans laquelle
  - Attend une seconde.
  - Incrémente le score de 1 et met à jour l'affichage.
  - Si le score atteint un multiple de 60 (secondes), il réduit le temps d'apparition des billes (qui doit donc être une variable globale) de 10%.
  - Remonte dans sa boucle.

Remarquez que lorsque le compteur de billes tombe à zéro, la partie est terminée et le score doit arrêter de s'incrémenter.

Etape 9 : Fin du jeu et fermeture de la fenêtre

Lorsque le compteur de billes tombe à 0, la partie est terminée mais la fenêtre graphique reste ouverte jusqu'au moment où le joueur cliquera sur la croix de la fenêtre. Dans cet intervalle de temps,

- Le score ne doit plus s'incrémenter.
- Les statues, les mages et le piston ne peuvent plus bouger et ne peuvent plus être mis en prison.
- De nouveaux murs aléatoires ne peuvent plus apparaître.

A vous de gérer cela comme vous voulez.

Remarques générales sur l'application

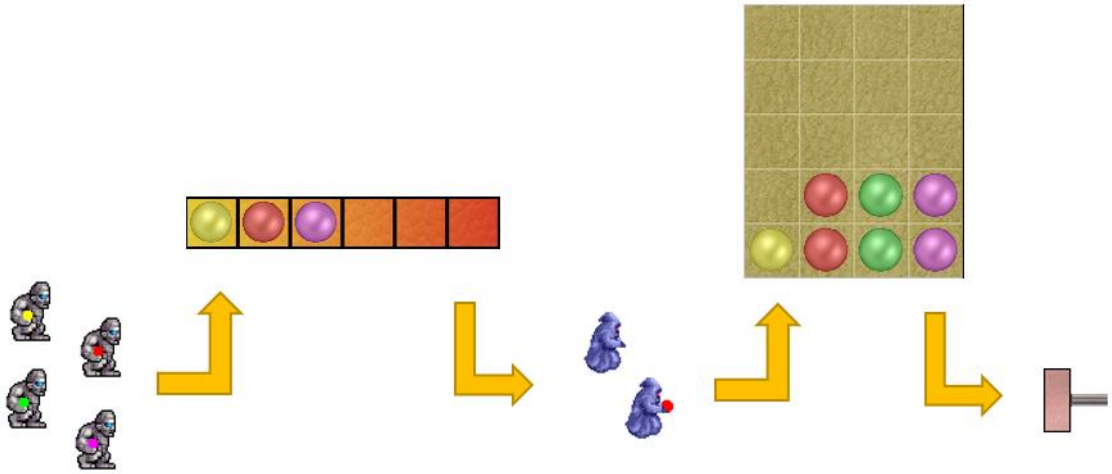
N'oubliez pas d'**armer** et de **masquer** correctement tous les **signaux** gérés par l'application !

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

Remarques concernant la théorie 😉

En rapport avec la théorie, vous avez implémenté plusieurs modèles (paradigmes) de programmation des threads :

- Vous avez tout d'abord implémenté le **modèle Producteur/Consommateur avec pool de threads** préalablement créés : il s'agit des threads Statues (consommateurs) et du thread Event (producteur)
- Vous avez implémenté une variante du **modèle pipeline** selon le schéma suivant :



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)/Thread2018**, 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 !