Laboratoire de Threads - Enoncé du dossier final
Année académique 2021-2022
Jeu de « Space Invader »

Il s'agit de créer un jeu du type « Space Invader » pour un joueur. La terre est attaquée par une flotte d'aliens qui descendent petit à petit vers la surface de la terre. Afin de défendre la planète assiégée, le joueur est responsable du déplacement et du tir de son vaisseau pouvant se déplacer latéralement à l'abri de boucliers ne résistant que partiellement. Le vaisseau lance des missiles vers les aliens afin de les détruire. Space Invader est un jeu sans fin. En effet, si la flotte aliens est entièrement détruite par les tirs de missiles, une nouvelle flotte est générée et, de plus, celle-ci se déplace plus rapidement que la précédente (ce qui provoque également le passage du jeu d'un niveau à un niveau supérieur). D'un autre côté, si la flotte aliens atteint la surface de la planète (représentée par les boucliers), l'invasion est réussie et le joueur perd une vie (retour au début du niveau en cours dans ce cas), le joueur disposant de 3 vies au total. De plus, les aliens bombardent la terre avec des bombes qui ont pour effet de détruire le vaisseau qui les reçoit. Le jeu s'arrête quand le joueur ne dispose plus de vie. Le but du jeu est d'accumuler le plus de points avant la fin inévitable de la terre ☹…

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 le clavier et la souris.

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

Notez dès à présent que plusieurs éléments sont déjà fournis :

- **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, vaisseau, aliens, missiles, bombes, 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 SpaceInvader.cpp qui contient déjà les bases de votre application et dans lequel vous verrez un exemple d'utilisation de la librairie GrilleSDL et 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 graphique 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. Chaque case de ce tableau contient un entier dont les valeurs possibles sont (macros déjà définies dans SpaceInvader.cpp) :

- 0 : VIDE
- 1 : VAISSEAU
- 2 : MISSILE
- 3 : ALIEN
- 4 : BOMBE
- 5 : BOUCLIER1 (bouclier état neuf)
- 6 : BOUCLIER2 (bouclier ayant été touché une fois par un missile ou une bombe)
- 7 : AMIRAL

mais également une variable de type pthread_t qui correspondant au « tid » (identifiant du thread) du thread qui occupe cette case (si la case est occupée par un thread, 0 sinon). Chaque case du tableau tab est donc une structure :

```
typedef struct {
  int        type;   // VIDE, VAISSEAU, MISSILE, …
  pthread_t  tid;    // identifiant du thread si case occupée par un thread
} S_CASE;
```

Le tableau tab et la librairie graphique fournie sont totalement indépendants. Dès lors, si vous voulez, par exemple, placer un missile à la case (7,15), c'est-à-dire à la ligne 7 et la colonne 15, vous devrez coder :

```
tab[7][15].type = MISSILE ;          // gère la logique de tout le jeu
tab[7][15].tid  = pthread_self() ;   // exécuté par un thread missile (voir plus loin)
DessineMissile(7,15) ;               // fonction du module Ressources,
                                     // pour dessiner dans la fenêtre graphique
```

tandis que pour effacer ce missile, il faut coder

```
tab[7][15].type = VIDE ;     // gère la logique de tout le jeu
tab[7][15].tid  = 0 ;        // Plus aucun thread à cette case
EffaceCarre(7,15) ;          // fonction du module GrilleSDL,
                             // pour effacer une case dans la fenêtre graphique
```

La fonction setTab(int l,int c,int type,pthread_t tid) vous est également fournie pour manipuler simultanément les champs type et tid d'une case du tableau.

Afin de contrôler son vaisseau, le joueur utilise les touches du clavier '←', '→' et 'espace' pour le faire aller à gauche ou à droite et tirer un missile respectivement. Le clic sur la croix de la fenêtre permet de terminer le jeu à tout moment.

Notez dès à présent que les boucliers ont deux codes différents selon qu'ils n'ont pas encore été touchés (code BOUCLIER1 et ligne verte sur le sprite) ou qu'ils ont été touchés une fois (code BOUCLIER2 et ligne rouge sur le sprite). Chaque bouclier doit en effet être touché deux fois (par une bombe ou un missile indistinctement) avant de disparaître. Pour leur affichage dans la fenêtre graphique, on utilisera la fonction DessineBouclier(int ligne, int colonne, int n) où le 3ème paramètre ne peut prendre que les valeurs 1 ou 2 selon que l'on veut dessiner un bouclier du type BOUCLIER1 ou BOUCLIER2. Remarquez enfin qu'un bouclier n'est pas un thread. Dès lors, une case occupée par un bouclier aura un type égal à BOUCLIER1 ou BOUCLIER2 et un tid égal à 0.

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 Vaisseau et d'un premier mutex

Dans un premier temps, le thread principal lancera le **threadVaisseau**. Le vaisseau ne peut se déplacer que latéralement, donc à gauche ou à droite sur la ligne du bas de la grille de jeu (d'indice NB_LIGNE – 1). Sa position est donc caractérisée par une variable globale **colonne** qui représente sa colonne actuelle dans la grille de jeu et qui peut prendre des valeurs allant de 8 à 22 (les colonnes allant de 0 à 7 correspondent à la zone grise d'affichage du logo, score, niveau et vies restantes). Le vaisseau démarrera sa course à la colonne 15.

Afin de dessiner le vaisseau dans la grille de jeu, il faudra utiliser la fonction DessineVaisseau(int ligne, int colonne). Le threadVaisseau est responsable de son propre affichage. Il doit donc mettre à jour le tableau **tab**, ainsi qu'appeler les fonctions d'affichage nécessaires à la fenêtre graphique.

Important ! Bien sûr, le vaisseau et autres intervenants du jeu ne peuvent pas se chevaucher… Vous devez donc dès maintenant protéger l'accès à la grille de jeu (à la variable tab donc) par un mutex : **mutexGrille**.

Une fois démarré et avoir positionné le vaisseau dans la grille de jeu (*il faut s'assurer que la case dans laquelle on veut introduire le vaisseau est libre !*), le threadVaisseau va se mettre en attente de signaux sur un **pause()**. La réception des signaux SIGUSR1, SIGUSR2 et SIGHUP lui indiqueront une demande pour aller à gauche, à droite, ou tirer un missile respectivement. La variable **colonne** étant globale, elle sera accessible dans les différents handlers de signaux.

Etape 2 : Création du threadEvent

Le thread principal va lancer le **threadEvent** dont le rôle est de gérer les événements provenant du clavier et 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 ». Dans le cas « croix de fenêtre », le threadEvent fermera la fenêtre graphique et terminera proprement le processus.

Le **threadEvent** attend donc 3 types d'événements provenant du clavier :

- Un clic sur la touche '←' (event.touche == KEY_LEFT) : cela doit avoir pour effet de faire déplacer le vaisseau d'une case vers la gauche : envoi d'un SIGUSR1 au threadVaisseau.
- Un clic sur la touche '→' (event.touche == KEY_RIGHT) : cela doit avoir pour effet de faire déplacer le vaisseau d'une case vers la droite : envoi d'un SIGUSR2 au threadVaisseau.
- Un clic sur la barre d'espace (event.touche == KEY_SPACE) : cela doit avoir pour effet de faire tirer un missile par le vaisseau : envoi d'un SIGHUP au threadVaisseau.

Dès réception de SIGUSR1 ou SIGUSR2, le threadVaisseau modifiera la variable **colonne** et le tableau **tab** en conséquence, et mettra à jour l'affichage dans la fenêtre graphique. Le vaisseau est donc maintenant capable de se déplacer par l'intermédiaire des touches du clavier mais il ne sait pas encore tirer de missile (réception de SIGHUP)… Allons-y…

Etape 3 : Création des threads Missile et du thread Time Out

Dès réception d'un SIGHUP, le threadVaisseau doit lancer un **threadMissile** avec un paramètre alloué dynamiquement (malloc) par le threadVaisseau. Ce paramètre est une « instance » de la structure suivante :

```
typedef struct {
  int L;
  int C;
} S_POSITION;
```

Les champs L et C seront initialisés par le threadVaisseau et correspondent à la position du missile sur la grille de jeu. Ils seront initialisés respectivement à la même valeur que la colonne du vaisseau au moment du tir (variable globale **colonne** pour l'instant) et à (NB_LIGNE-2).

Une fois démarré, le threadMissile doit dans un premier temps initialiser les champs type et tid de la case sur laquelle il démarre. Mais attention, il doit d'abord vérifier s'il peut entrer dans la grille de jeu ! Actuellement, 3 cas d'entrée dans la grille sont possibles :

1. Il est lancé sur une case vide, dans ce cas, pas de souci…
2. Il est lancé sur une case contenant BOUCLIER1, dans ce cas, la case devient BOUCLIER2 et le threadMissile doit se terminer.
3. Il est lancé sur une case contenant BOUCLIER2, dans ce cas, la case devient VIDE et le threadMissile doit se terminer.

Une fois entré dans la grille de jeu, il vit sa vie en avançant d'une case vers le haut toutes les **80 millisecondes** (la fonction **Attente(int milli)** vous est fournie). Une fois arrivé, tout en haut de la grille de jeu, le threadMissile doit se terminer.

Bon, on avance ☺ ! Mais le tir du vaisseau ressemble plus à un tir de rayon laser qu'à un tir de missiles ☹. En effet, l'appui rapide et répété sur la barre d'espace provoque la création d'un nombre très important de missiles… On va limiter cela en n'autorisant un tir de missile que toutes les **600 millisecondes** au maximum. Pour cela, au lancement de chaque threadMissile, le threadVaisseau devra mettre une variable globale **fireOn** (de type bool initialement mise à true) à false et lancer un **threadTimeOut** dont le rôle est de :

- faire une attente de 600 millisecondes,
- remettre la variable globale **fireOn** à true.
- se terminer.

Très bien, on peut se défendre maintenant ! Mais personne ne nous attaque… On y vient ☺.

Etape 4 : Création du thread « Invader » et du thread Flotte Aliens

A partir du thread principal, lancer le **threadInvader** dont le rôle est de gérer l'invasion de la terre, ainsi que le niveau du jeu (ainsi que son affichage dans la fenêtre graphique). Ce thread tourne en permanence et réalise les opérations suivantes :

- il crée le **threadFlotteAliens**,
- il se synchronise sur la fin du threadFlotteAliens (utilisation de **pthread_join**),
- une fois le threadFlotteAliens terminé, deux cas sont possibles :
  - soit la flotte aliens a été complètement détruite par les tirs de missiles,
  - soit la flotte aliens a atteint les boucliers terrestres et l'invasion est réussie.

Nous reviendrons en détails sur ces deux cas après avoir décrit le threadFlotteAliens.

Le **threadFlotteAliens** est responsable du déplacement de l'ensemble des aliens de la flotte, ainsi que de leur interaction avec les autres intervenants du jeu (missiles, bombes à venir). Il n'y donc pas un thread par alien mais bien un seul thread pour toute la flotte !!!

Au démarrage, le **threadFlotteAliens** doit commencer par placer **24** aliens sur la grille de jeu. Ces 24 aliens sont disposés sur 4 lignes comportant 6 aliens chacune. Les aliens doivent être séparés par une case vide aussi bien entre colonne qu'entre ligne. Le premier alien (en haut à gauche) aura donc une position initiale (ligne,colonne) = (2,8) tandis que le dernier (en bas à droite) aura une position initiale (ligne,colonne) = (8,18). La flotte aliens sera caractérisée par 5 variables globales **nbAliens** (nombre d'aliens présents sur la grille), ainsi que **lh, cg, lb, cd** qui représentent respectivement la ligne de l'alien le plus haut de la flotte, la colonne de l'alien le plus à gauche de la flotte, la ligne de l'alien le plus bas de la flotte et la colonne de l'alien le plus à droite de la flotte. Ces 4 paramètres définissent donc le plus petit « cadre » englobant la flotte aliens. Ce cadre permettra de connaître les limites de déplacement de la flotte sur la grille, les aliens ne pouvant sortir de la grille. Au départ, ces 5 variables globales auront les valeurs **nbAliens=24, lh=2, cg=8, lb=8 et cd=18**. Ces paramètres seront modifiés par le threadFlotteAliens lorsque celle-ci se déplacera mais également par les threads missiles. Elles doivent donc être protégées par un **mutexFlotteAliens**.

La flotte se déplace d'une case toutes les secondes (mais ce laps de temps diminuera au fur et à mesure que l'on change de niveau) (utilisation de **Attente**). Ce délai d'attente (1000 millisecondes) doit être stocké dans une variable globale **delai**. La flotte commence par un aller-retour complet droite-gauche. Une fois l'aller-retour terminé, la flotte descend d'une case et un nouvel aller-retour droite-gauche est réalisé. Et ainsi de suite jusqu'au moment où la flotte aliens ne sait plus descendre car elle a atteint la ligne juste au-dessus des boucliers. Dans ce cas, le threadFlotteAliens se termine.

A chaque déplacement, le **threadFlotteAliens** doit vérifier la variable **nbAliens** (qui peut être modifiée entre temps par un thread missile qui aurait détruit un alien). Si celle-ci est à zéro, la flotte a été détruite et le threadFlotteAliens se termine.

Passons un instant sur l'interaction « missile-alien ». Au moment de son déplacement, le **threadFlotteAliens** doit vérifier, pour chaque alien, que celui-ci ne se positionne pas sur un missile. Dans ce cas, le threadFlotteAliens doit décrémenter nbAliens (ne pas oublier de mettre à jour lh,lb,cg,cd !) lui-même, mettre le tableau tab à jour en conséquence et également prévenir le threadMissile que celui-ci a touché un alien et doit être détruit. Le tid du missile étant présent dans la case qu'il occupe, il sera possible de lui envoyer le signal SIGINT. Le handlerSIGINT(int s) devra simplement contenir un appel à la fonction **pthread_exit**. Chaque threadMissile doit donc être configuré pour recevoir le signal SIGINT.

D'un autre côté, chaque missile, lorsqu'il a la main, doit maintenant gérer le cas de la rencontre avec un alien. Dans ce cas, le threadMissile modifie tab, nbAliens, cg, cd, lh, lb avant de se terminer.

Nous pouvons à présent revenir sur le **threadInvader**. Lorsque le threadFlotteAliens se termine, le threadInvader peut réagir de 2 manières possibles :

- **Soit nbAliens est égal à 0.** Dans ce cas, l'invasion a échoué. Il faut alors lancer un nouveau threadFlotteAliens, mais la nouvelle flotte doit se déplacer plus rapidement, ce qui augmente la difficulté du jeu. Pour cela, avant de relancer le threadFlotteAliens, le thread Invader diminue la valeur de la variable globale delai de 30% (ceci aura donc pour effet que les aliens se déplaceront plus rapidement d'un niveau à l'autre). De plus, il doit augmenter le niveau de 1 dans la fenêtre graphique aux cases (13,3) et (13,4). Pour cela, il dispose de la fonction DessineChiffre(int l,int c,int valeur).

- **Soit nbAliens est différent de 0.** Dans ce cas, l'invasion est réussie. Le threadFlotteAliens doit alors envoyer un signal SIGQUIT au threadVaisseau afin de le détruire. Donc, le threadVaisseau doit à présent gérer le signal SIGQUIT qui a pour effet de le faire disparaître de la grille avant de se terminer par un pthread_exit.

Dans les 2 cas, le **threadInvader** doit reconstruire les boucliers (intacts) et supprimer les aliens qui restent (s'il en reste bien sûr) avant de relancer une nouvelle flotte aliens.

Etape 5 : Création du thread Score

A partir du thread principal, lancer le **threadScore** dont le rôle est d'afficher le **score** dans la partie gauche de la fenêtre graphique.

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 (10,2), (10,3), (10,4) et (10,5) 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 threadMissile :

A chaque fois qu'un threadMissile rencontre et détruit un alien, celui doit incrémenter le score de 1 et réveiller le threadScore (**pthread_cond_signal**).

Retour sur le threadFlotteAliens :

De la même manière, le threadFlotteAliens doit également augmenter le score de 1 lorsqu'un alien se positionne sur un missile, avant de réveiller le threadScore.

Bon… Il commence à y avoir de la lutte mais le jeu est un peu trop simple, les aliens descendent mais ne tirent rien, on peut les tirer comment des lapins. Rendons-les plus agressifs ☺.

Etape 6 : Création des threads Bombe

On va donner la possibilité aux aliens de lâcher des bombes qui ont pour effet de détériorer/détruire les boucliers de protection ou de détruire le vaisseau.

Pour cela, un déplacement sur deux, le threadFlotteAlien choisit au hasard un des aliens restant dans la flotte. Une fois le déplacement réalisé, l'alien choisi lance un **threadBombe** avec un paramètre alloué dynamiquement. Ce paramètre est une « instance » de la structure S_POSITION déjà utilisée pour le lancement des missiles. Le threadFlotteAlien initialise les champs L et C de la structure de telle sorte que la bombe apparaisse juste en-dessous de l'alien qui a été choisi au hasard.

Une fois démarré, le **threadBombe** doit dans un premier temps initialiser les champs type et tid de la case sur laquelle il démarre. Comme pour les missiles, mais surtout pour les bombes, l'entrée sur la grille de jeu doit être particulièrement soignée. En effet, il se peut qu'une bombe soit lancée sur un missile, un bouclier du type BOUCLIER1 ou BOUCLIER2, voir même sur une autre bombe. Dans le cas d'un missile, celui-ci sera détruit par l'envoi d'un SIGINT. Dans le cas d'une bombe, le thread se terminera sans rien faire.

Une fois entrée sur la grille de jeu, la bombe descend d'une ligne toutes les **160 millisecondes**, et ce jusqu'à la ligne d'indice NB_LIGNE-1. Le threadBombe se termine alors.

En cours de route, le **threadBombe** peut rencontrer

- un missile, ce qui pour effet de le détruire en lui envoyant un SIGINT. Le threadBombe se termine également. Le tid du missile aura été obtenu dans le tableau tab à la case correspondante.
- un bouclier du type BOUCLIER1. Dans ce cas, le bouclier devient du type BOUCLIER2 et le threadBombe disparait.
- un bouclier du type BOUCLIER2. Dans ce cas, le bouclier disparait et la case prend la valeur VIDE. Le threadBombe se termine alors.
- le vaisseau, dans ce cas le threadBombe se termine et le vaisseau doit être détruit. Pour cela, il suffit de lui envoyer le signal SIGQUIT qui est déjà géré par le threadVaisseau.
- un alien, dans ce cas, la threadBombe passe son tour et attend que l'alien se soit déplacé.

Retour sur le threadMissile :

Il faut à présent également modifier le code du threadMissile pour qu'il tienne compte de sa rencontre avec une bombe.

Retour sur le threadFlotteAliens :

Mais que se passe-t-il lorsqu'un alien, en se déplaçant, doit se positionner sur une bombe ? Dans ce cas, le threadFlotteAliens doit supprimer cette bombe en envoyant un signal SIGINT au threadBombe correspondant puis relancer une nouvelle bombe en dessous de l'alien concerné.

Etape 7 : Gestion des vies et synchronisation du thread principal

Le vaisseau peut à présent être détruit, et ce, de plusieurs manières, soit l'invasion est terminée, soit il a été détruit suite à la réception d'une bombe. Il est maintenant temps de gérer le nombre de vies et la fin de partie.

Après avoir lancé les différents threads (Event, Score, Invader), et après avoir lancé un premier threadVaisseau, le thread principal doit se mettre en attente de la réalisation de la condition (à l'aide d'un mutex **mutexVies** et d'une variable de condition **condVies**) suivante :

« Tant que (nbVies) > 0, j'attends… »

où **nbVies** est une variable globale représentant le nombre de vies restant au joueur. Cette variable sera initialisée à 3 en début de partie. Une fois réveillé et ayant vérifié qu'il reste assez de vies au joueur, il relancera un nouveau threadVaisseau.

Lorsque la condition n'est plus vérifiée, le thread principal affiche « Game Over » dans la fenêtre graphique en utilisant la fonction DessineGameOver(6,11). La fin du processus est alors gérée par le threadEvent qui attend que l'utilisateur clique sur la croix de la fenêtre.

C'est évidemment au threadVaisseau que revient la tâche de décrémenter la variable **nbVies**. Pour cela, on demande de mettre en place, pour le threadVaisseau, une fonction de terminaison (utilisation de **pthread_cleanu_push** et **pthread_cleanup_pop**).

Etape 8 : Création du thread Vaisseau Amiral

Lorsqu'un certain nombre d'aliens ont été détruits (disons lorsque la variable globale **nbAliens** devient multiple de 6), le vaisseau amiral alien apparaît afin de se rendre compte de la situation. Ce vaisseau occupe 2 cases sur la grille de jeu et passe rapidement sur la ligne supérieure (indice 0) et ne tire pas de bombe. Il apparaît (utilisation de la fonction **DessineVaisseauAmiral (int l,int c)** où (l,c) correspond à la case de gauche du vaisseau) pour un certain temps aléatoire avant de disparaître à nouveau. Un tir de missile peut le toucher, ce qui attribue 10 points au joueur qui l'a touché, mais il n'est jamais détruit.

Pour ce faire, à partir du thread principal, lancer le **threadVaisseauAmiral** (thread qui tourne en permanence) réalisant les actions suivantes :

1. il se met en attente de la réalisation de la condition (à l'aide du mutex **mutexFlotteAliens** (existant déjà) et de la variable de condition **condFlotteAliens**) suivante :
   « Tant que (nbAliens%6) != 0) OU (nbAliens==0)), j'attends… »

2. Dès que la condition est réalisée (nbAliens est multiple de 6 et nbAliens différent de 0), il choisit une position libre (il faut 2 cases libres !) au hasard sur la ligne d'indice 0 de la grille, positionne le vaisseau amiral sur la grille et choisit une direction de déplacement aléatoire sur cette ligne (gauche ou droite).

3. Il va demander à recevoir un **signal SIGALRM** après un temps aléatoire compris entre 4 et 12 secondes (utilisation de **alarm**). Attention, seul le threadVaisseauAmiral peut recevoir le SIGALRM !

4. Il va se déplacer d'une case toutes les **200 millisecondes** jusqu'au moment où le signal SIGALRM sera reçu ou qu'il aura été touché par un missile. La réception du SIGALRM provoque la disparition du vaisseau de la grille. Dans le cas où le vaisseau amiral rencontre un missile lors de son déplacement, il doit s'arrêter le temps de laisser passer le missile. Par contre, si un missile, lors de son déplacement, rencontre le vaisseau amiral, il lui enverra le signal SIGCHLD afin qu'il puisse annuler l'alarme (utilisation de **alarm(0)**) et disparaître.

5. Il remonte dans sa bouche afin de se remettre en attente sur la variable de condition.

Etape 9 (BONUS NON OBLIGATOIRE) : Threads Bouclier et variable spécifique

Les boucliers deviennent des threads. Un thread doit être lancé pour chaque bouclier. Chaque **threadBouclier** doit disposer d'une **variable spécifique** (utilisation de **pthread_setspecific** et **pthread_getspecific**) **entière allouée dynamiquement** et représentant sont état (1 intact ou 2 abîmé). Lorsqu'un missile ou une bombe atteint un bouclier, un signal (celui de votre choix) doit être envoyé au threadBouclier correspondant. Dans le handler de signal, le threadBouclier mettra son état (sa variable spécifique donc) à jour et disparaîtra de la grille de jeu s'il est touché une seconde fois. Une fois le bouclier disparu, le threadBouclier doit se terminer.

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
threadVaisseau	SIGUSR1, SIGUSR2, SIGHUP, SIGQUIT
threadMissile	SIGINT
threadTimeOut	Aucun
threadFlotteAliens	Aucun
threadInvader	Aucun
threadScore	Aucun
threadBombe	SIGINT
threadVaisseauAmiral	SIGALRM, SIGCHLD

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

Mutex	Variables globales
mutexGrille	tab
mutexFlotteAliens	nbAliens, cg, cd, lh, lb
mutexScore	score, MAJScore
mutexVies	nbVies

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 la machine Jupiter de l'école** 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 votre répertoire **$(HOME)/Thread2022**, 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 😊 !