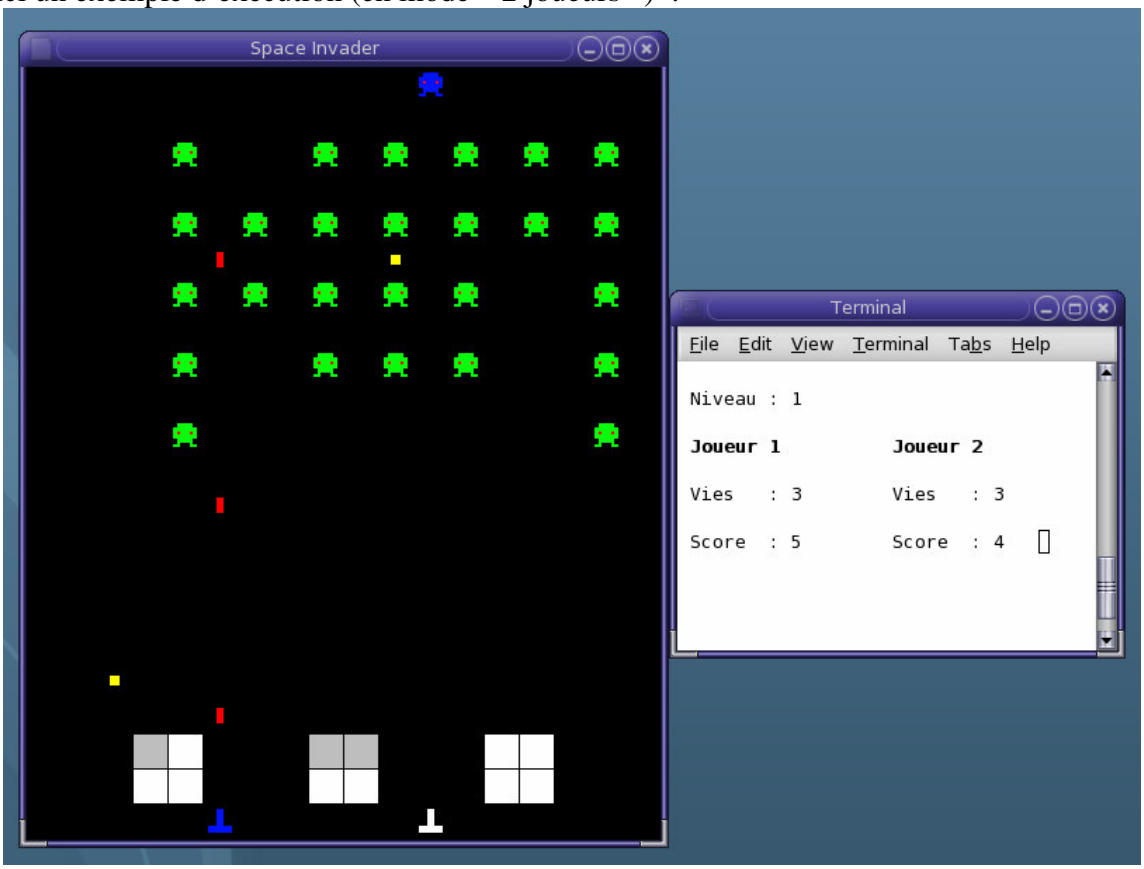
Laboratoire de Threads - Enoncé du dossier final
Année académique 2010-2011
Jeu de « Space Invader »

Il s'agit de créer un jeu du type « Space Invader » pour 1 ou 2 joueur(s). 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, chaque joueur est responsable du déplacement et du tir de son propre canon pouvant se déplacer latéralement à l'abri de deux boucliers ne résistant que partiellement. Les canons lancent 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 canon, 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 les deux joueurs perdent une vie chacun (retour au début du niveau en cours dans ce cas), chaque joueur disposant de 3 vies au total. De plus, les aliens bombardent la terre avec des bombes qui ont pour effet de détruire le canon qui les reçoit. Le jeu s'arrête quand plus aucun des 2 joueurs ne disposent de vie, un joueur pouvant continuer seul la partie si l'autre joueur a perdu toutes ses vies. 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. Une fenêtre graphique apparaîtra alors. C'est dans cette fenêtre que devra apparaître la grille de jeu. L'affichage des scores sera assuré en mode texte dans le terminal qui a lancé l'application.

Voici un exemple d'exécution (en mode « 2 joueurs ») :



Notez dès à présent que plusieurs librairies vous sont fournies dans le répertoire **/export/home/public/wagner/EnonceThread2011**. Il s'agit de

- Ecran : librairie que vous connaissez déjà bien. En particulier, la fonction AffChaine permet d'afficher une chaine de caractères à un endroit précis du terminal.
- EcranX : une librairie simplifiée de terminal graphique (basée sur Xlib).
- Grille : une librairie 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, un canon, un missile, un morceau de bouclier, une bombe, un alien, … Ces fonctions sont décrites en détails ci-dessous.

De plus, vous trouverez le fichier SpaceInvader.c qui contient déjà les bases de votre application (dessin de la grille et des boucliers) et dans lequel vous verrez un exemple d'utilisation de la librairie Grille. Vous ne devez donc en aucune façon programmer la moindre fonction qui a un lien avec le terminal graphique. Vous ne devrez accéder au terminal graphique que via les fonctions de la librairie Grille. Vous devez donc vous concentrer uniquement sur la programmation des threads !

Les choses étant dites, venons-en aux détails de l'application… La grille de jeu sera représentée par un tableau à 2 dimensions (variable **tab**) défini en global. Ce tableau contient des entiers dont le code (macros déjà définies dans SpaceInvader.c) correspond à

- 0 : VIDE
- 1 : CANON1 (celui du joueur 1)
- 2 : CANON2 (celui du joueur 2)
- 3 : MISSILE
- 4 : BOUCLIER1 (bouclier état neuf)
- 5 : BOUCLIER2 (bouclier ayant été touché une fois par un missile ou une bombe)
- 6 : ALIEN
- 7 : BOMBE
- 8 : VAISSEAU_AMIRAL

Le canon du joueur 1 (unique dans le cas du mode « 1 joueur ») aura une couleur blanche et sera représenté par un « T » retourné. Il en est de même pour le canon du joueur 2 mis à part que sa couleur est bleue.

Afin de contrôler leur canon, les joueurs utilisent les touches du clavier :

- Joueur 1 (canon blanc) : touches '←', →' et 'espace' pour le faire aller à gauche ou à droite et tirer un missile respectivement.
- Joueur 2 (canon bleu) : touches 'w' , 'x' et 's' pour le faire aller à gauche ou à droite et tirer un missile respectivement.

Les macros GAUCHE et DROITE vous sont d'ailleurs fournies.

Notez dès à présent que les boucliers ont deux codes différents selon qu'ils n'ont pas encore été touché (code BOUCLIER1 et couleur blanche) ou qu'ils ont été touchés une fois (code BOUCLIER2 et couleur gris foncé). Chacune des 4 cases constituant un bouclier doit en effet être touchée deux fois (par une bombe ou un missile indistinctement) avant de disparaître. Pour leur affichage dans le terminal 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.

Le terminal graphique ne servant qu'à l'affichage de la grille de jeu, c'est le terminal console qui gère l'application et donc le clavier (à l'aide de la fonction **ReadChar** qui vous est fournie dans SpaceInvader.c).

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**.

Les 7 premières étapes ne concernent que le mode « 1 joueur ». Le mode « 2 joueurs » sera expliqué à l'étape 8. L'étape 9 constitue un bonus pour ce dossier.

Etape 1 : Création du thread Canon

Dans un premier temps, le thread principal ne lancera qu'un seul **threadCanon** (mode 1 joueur donc). Un canon 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 **colonne** qui représente sa colonne actuelle dans la grille de jeu.

Afin de dessiner le canon dans la grille de jeu, il faudra utiliser la fonction DessineCanon(int ligne, int colonne, int joueur) dont le 3ème paramètre ne pourra prendre que les valeurs 1 ou 2 selon que l'on veut dessiner la canon du joueur 1 ou 2. Le threadCanon 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, les canons 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 canon dans la grille de jeu (*il faut s'assurer que la case dans laquelle on veut introduire le canon est libre !*), le threadCanon 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. Pour cela, vous mettrez (DANS UN PREMIER TEMPS SEULEMENT) la variable **colonne** en globale afin qu'elle soit accessible dans les différents handlers de signaux ; ce qui ne pose pas de problème pour l'instant car il n'y a qu'un seul canon.

Etape 2 : Création du thread Clavier

A partir du thread principal, lancer le **threadClavier** qui sera responsable de la gestion du clavier. Pour cela il utilisera la fonction ReadChar() qui est déjà fournie dans SpaceInvader.c.
Ce thread doit réagir à la saisie des touches suivantes :

- 'espace' ('s' pour le second canon) : cela a pour effet de faire tirer un missile par le canon : envoi d'un SIGHUP au threadCanon correspondant.
- '←' ('w' pour le second canon) : cela a pour effet de faire déplacer le canon d'une case vers la gauche : envoi d'un SIGUSR1 au threadCanon correspondant.

- '→' ('x' pour le second canon) : cela a pour effet de faire déplacer le canon d'une case vers la droite : envoi d'un SIGUSR2 au threadCanon correspondant.

Dès réception de SIGUSR1 ou SIGUSR2, le threadCanon modifiera la variable **colonne** et le tableau **tab** en conséquence, et mettra à jour l'affichage dans la fenêtre graphique. Remarquez à nouveau que, vu que la variable colonne est globale pour l'instant, elle est accessible dans les handlers de signaux. Le canon 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 (rouge) et du thread Time Out

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

```
typedef struct s_missile
{
   int colonne;
   int ligne;
   int joueur;
   pthread_t tid ;
   struct s_missile *suivant ;
} S_MISSILE ;
```

Les 3 premiers champs seront initialisés par le threadCanon, les 2 derniers par le threadMissile créé (voir ci-dessous). Les champs (colonne,ligne) correspondent à la position du missile sur la grille de jeu et seront initialisés respectivement à la même valeur que la colonne du canon au moment du tir (variable globale **colonne** pour l'instant) et à (NB_LIGNE-2). Le champ joueur n'aura réellement de sens que lorsqu'on parlera du mode « 2 joueurs », actuellement il sera initialisé à 1.

Une fois démarré, le threadMissile doit terminer d'initialiser la structure qu'il a reçue en paramètre. Il doit dans un premier temps initialiser le champ tid à son propre tid (utilisation de **pthread_self()**). Les missiles devront être accessibles en permanence par les autres intervenants du jeu, comme les bombes ou les aliens. Dès lors, les structures S_MISSILE devront être stockées en global dans une liste liée dont le pointeur de tête **pListeMissiles** (du type S_MISSILE*) sera déclaré en global. Plusieurs fonctions d'accès à cette liste devront être implémentées :

- **void insereMissile(S_MISSILE *pm)** : dont le rôle est d'insérer la structure passée en paramètre dans la liste liée. Le champ suivant de la structure a donc été introduit dans un but de chainage des missiles dans la liste.
- **pthread_t getTidMissile(int l,int c)** : qui permet d'obtenir le tid du thread missile se situant à la position (l,c) de la grille de jeu.
- **S_MISSILE* getMissile(int l,int c)** : qui permet d'obtenir l'adresse de la structure associée au missile se situant à la position (l,c) de la grille de jeu.
- **S_MISSILE* retireMissile(S_MISSILE *pm)** : qui retire la structure passée en paramètre de la liste chainée tout en assurant la cohérence du chainage.

Bien sûr, cette liste liée étant globale, son accès devra être protégé par le **mutexListeMissiles** dont le lock() et le unlock() devront encadrer tout appel à ces fonctions.

Donc, une fois lancé, le threadMissile fera appel à la fonction insereMissile afin de s'insérer dans la liste des missiles. Il pourra ensuite entrer sur la grille de jeu. Actuellement, 3 cas d'entrée sur la grille sont possibles :

1. Il est lancé sur une case vide, dans ce cas, pas de souci…
2. Il est lancé sur un 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 rentré, il vit sa vie en avançant d'une case vers le haut toutes les 0,08 secondes (80.000.000 ns) (utilisation de **nanosleep**). Une fois arrivé, tout en haut de la grille de jeu, le threadMissile doit se terminer.

Pour la fin du threadMissile, il ne faut pas oublier de libérer la mémoire allouée pour la structure S_MISSILE correspondant. Pour cela, on vous demande d'utiliser une fonction de terminaison (utilisation de **pthread_cleanup_push** et **pthread_cleanup_pop**) dont le rôle sera de retirer la structure de la liste liée (appel à retireMissile) et d'en faire le free.

Bon, on avance ☺ ! Mais le tir du canon ressemble plus à un tir de rayon laser qu'à un tir de missiles ☹. En effet, l'appui en continu 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 0,6 seconde au maximum. Pour cela, au lancement de chaque threadMissile, le threadCanon devra mettre une variable **fireOn** (initialement mise à 1) à 0 (variable qui doit être déclarée en globale dans un premier temps uniquement !) et lancer un **threadTimeOut** dont le rôle est de :

- faire un nanosleep de 0,6 seconde,
- d'envoyer un signal SIGCONT au threadCanon pour le prévenir qu'il peut réactiver son tir. Le tid du thread canon aura été reçu en paramètre au lancement du thread.
- se terminer.

Dès lors, la threadCanon doit maintenant, en plus des signaux SIGUSR1, SIGUSR2 et SIGHUP, gérer le signal SIGCONT. Dès réception de ce signal, il remettra la variable **fireOn** à 1. Bien sûr, le tir d'un missile n'est autorisé que si la variable fireOn est à 1.

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 sur la console). Ce thread tourne en permanence et réalise les opérations suivantes :

- crée le **threadFlotteAliens**,
- 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 canon, 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 **35** aliens sur la grille de jeu. Ces 35 aliens sont disposés sur 5 lignes comportant 7 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) = (1,0) tandis que le dernier (en bas à droite) aura une position initiale (ligne,colonne) = (9,12). 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 de 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=35, lh=1, cg=0, lb=9 et cd=12**. Ces paramètres seront modifiés par le threadFlotteAliens lorsque celle-ci se déplacera mais également par les thread 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 **nanosleep**). 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. La fonction **getTidMissile** permettra au threadFlotteAlien d'obtenir le tid du missile ayant fait mouche et de lui envoyer le signal SIGINT. Le handlerSIGINT(int s) devra simplement contenir un appel à la fonction **pthread_exit** (la fonction de terminaison du threadMissile sera alors automatiquement appelée). 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 relancer une nouveau threadFlotteAliens, mais la nouvelle flotte doit se déplacer plus rapidement, ce qui augmente la difficulté du jeu. Pour cela, le threadFlotteAliens est lancé avec un paramètre du type **struct timespec**. A

chaque fois qu'une flotte aliens est détruite, le niveau augmente de 1 tandis que le délai entre deux déplacements de la flotte aliens est réduit de 30% de sa valeur. Le délai entre deux déplacements de la flotte aliens au niveau 1 est, pour rappel, de 1 seconde.

- **Soit nbAliens est différent de 0.** Dans ce cas, l'invasion est réussie. Le threadFlotteAliens doit alors envoyer un signal SIGQUIT au threadCanon afin de le détruire. Donc, le threadCanon 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 deux boucliers 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(s) score(s) dans la console. Le score du joueur 1 est représenté par la variable globale **score1** qui sera protégée par le **mutexScore**. Le threadScore se mettra donc attente sur la variable de condition **condScore** combinée au mutexScore (utilisation de **pthread_cond_wait**). A chaque réveil, le threadScore met à jour l'affichage du score.

Mais qui va modifier la variable score1 ? Deux possibilités. Tout d'abord, il peut s'agir d'un thread missile qui vient de détruire un alien. Réciproquement, le threadFlotteAliens peut également modifier score1 lorsqu'un alien se positionne sur un missile. Après modification « atomique » de la variable score1, le thread concerné devra réveiller le threadScore par un **pthread_cond_signal**. Pour tout alien détruit, un point est attribué au joueur.

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 (jaune)

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 canon.

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 suivante

```
typedef struct s_bombe
{
   int colonne;
   int ligne;
   pthread_t tid ;
   struct s_bombe *suivant ;
} S_BOMBE ;
```

Le threadFlotteAlien initialise les champs colonne et ligne de la bombe de telle sorte qu'elle apparaisse juste en-dessous de l'alien qui a été choisi au hazard.

Les bombes vont être gérées exactement de la même manière que les missiles. Vous devez donc introduire une variable globale **pListeBombes** qui représente le pointeur de tête de la liste chainée de bombes. De plus, cette liste sera gérée par les 4 fonctions suivantes

- **void insereBombe(S_BOMBE *pb)**
- **pthread_t getTidBombe(int l,int c)**
- **S_BOMBE* getBombe(int l,int c)**
- **S_BOMBE* retireBombe(S_BOMBE *pb)**

que vous devez implémenter et qui sont tout à fait similaires à celles concernant la liste de missiles. L'accès à cette liste devra être protégé par le **mutexListeBombes**.

Une fois lancé, le **threadBombe** doit initialiser le champ tid à son propre tid et insérer la structure reçue en paramètre dans la liste des bombes en utilisant la fonction **insereBombe**.

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 0,16 seconde (utilisation de **nanosleep**), et ce jusqu'à la ligne d'indice NB_LIGNE-1. Le threadBombe se termine alors en appelant une fonction de terminaison (**pthread_cleanup_push** et **pthread_cleanup_pop**) qui le supprime de la liste des bombes (utilisation de **retireBombe**) et en fait le free.
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 par un appel à getTidMissile.
- 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.
- un canon, dans ce cas le threadBombe se termine et le canon doit être détruit. Pour cela, il suffit de lui envoyer le signal SIGQUIT qui est déjà géré par le threadCanon.
- un alien, dans ce cas, la threadBombe passe son tour et attend que l'alien se soit déplacé.

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

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 canon peut à présent être détruit, et ce, de plusieurs manière, 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 (Clavier, Score, Invader), et après avoir lancé un premier threadCanon, 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 (nbVies1) > 0, j'attends… »

où **nbVies1** est une variable globale représentant le nombre de vies restant au joueur 1. 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 threadCanon. Lorsque la condition n'est plus vérifiée, le thread principal s'arrête provoquant la fermeture de la grille de jeu et l'arrêt du processus.

C'est évidemment au threadCanon que revient la tâche de décrémenter la variable **nbVies1** et de réveiller le thread principal lorsqu'il passe par sa fonction de terminaison.

Etape 8 : Gestion d'un deuxième canon (mode « 2 joueurs »)

Avant toute chose, au lancement de l'application, le programme doit demander à l'utilisateur le nombre de joueurs. L'ajout d'un 2ème joueur ne va pas changer beaucoup de chose par rapport à ce qui a déjà été fait.

Au lieu de lancer un threadCanon avant de se mettre en attente sur la variable de condition, le thread principal lancera les deux threads canons des deux joueurs (**threadCanon1** et **threadCanon2**).
Le threadClavier reste inchangé, il a en effet déjà été configuré pour gérer les deux canons.

Le gros souci se situe au niveau des variables **colonne** et **fireOn**. Elles ne peuvent plus être globales… On vous demande donc de créer une **variable spécifique** (utilisation de **pthread_setspecific** et **pthread_getspecific**) par thread canon et ayant la structure suivante :

```
typedef
{
      int  colonne ;
      char fireOn ;
      int  joueur ;
} S_CANON ;
```

et reprenant toutes les caractéristiques d'un canon. Cette variable sera donc accessible par toutes les fonctions utilisées par le threadCanon comme sa fonction de terminaison ou les handlers de signaux (utilisation de **pthread_getspecific**)… ☺

Les threads canon seront alors lancés en recevant en paramètre une structure de ce type allouée dynamiquement par le thread principal. Les canons devront apparaître sur des colonnes différentes. Le champ **joueur** sera initialisé à 1 ou 2 selon le joueur. Une fois lancé,

chaque threadCanon placera l'adresse de la structure dans sa zone spécifique (utilisation de **pthread_setspecific**).

Pour la gestion des scores, il doit maintenant y avoir 2 variables globales **score1** et **score2** représentant le score de chacun des joueurs. Mais comment un missile qui vient de détruire un alien peut-il savoir à quel joueur attribuer le point ? C'est maintenant que le **champ joueur de la structure S_MISSILE** prend toute son importance. Un missile appartient à un joueur et le champ joueur de la structure S_MISSILE doit être assigné à 1 ou 2 selon que c'est le canon du joueur 1 ou du joueur 2 qui tire.

Bien sûr, c'est toujours le **threadScore** qui est responsable de l'affichage des scores et le **mutexScore** protègera à présent **score1** et **score2**.

Reste maintenant à modifier le thread principal pour la gestion de la fin de partie. Donc, dans le mode « 2 joueurs », le déroulement des opérations est le suivant. Après avoir lancé les différents threads (Clavier, Score, Invader), et après avoir lancé un premier threadCanon pour chaque joueur, le thread principal doit se mettre en attente de la réalisation de la condition (à l'aide du mutex **mutexVies** et de la variable de condition **condVies**) suivante :

« Tant que ((nbVies1 > 0) OU (nbVies2) > 0)), j'attends… »

où **nbVies1** et **nbVies2** sont 2 variables globales représentant le nombre de vies restant au joueur 1 et au joueur 2. Ces variables seront initialisées à 3 en début de partie. Une fois réveillé et ayant vérifié qu'il reste assez de vies au(x) joueur(s), il relancera un nouveau threadCanon au(x) joueur(s) qui ont perdu une vie. Mais attention donc !!! Il se peut qu'un des 2 joueurs perde une vie et pas l'autre. Il ne faut pas relancer un threadCanon pour le joueur qui n'a pas perdu de vie ! Pour cela, vous devez utiliser 2 variables globales **enVie1** et **enVie2** qui reste à 1 tant que le canon du joueur 1 ou du joueur 2 n'est pas détruit. Le test de ces variables permettra au thread principal de savoir s'il doit relancer un threadCanon pour tel ou tel joueur ou les deux. C'est le **mutexVies** qui protègera l'accès aux variables globales **nbVies1**, **nbVies2**, **enVie1** et **enVie2**.

Remarquez que dans le cas où l'invasion de la flotte aliens est réussie, les deux threads canons doivent être détruits par l'envoi d'un SIGQUIT par le **threadInvader**.

Etape 9 (BONUS) : 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 passe rapidement sur la ligne supérieure (indice 0) et ne tire pas de bombe. Il apparaît (sous la forme d'un alien de couleur bleue) pour un certain temps aléatoire avant de disparaître à nouveau. Un tir de canon 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)), j'attends… »

2. Dès que la condition est réalisée (nbAliens est multiple de 6), il choisit une position libre 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 0,2 seconde 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 faire un appel à alarm(0) pour annuler l'alarme et envoyer le signal SIGINT au threadMissile correspondant afin de le détruire. Réciproquement, si un missile, lors de son déplacement, rencontre le vaisseau amiral, il lui enverra le signal SIGCHLD afin qu'il puisse annuler l'alarme.

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

Remarques

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 peut donc que vous deviez ajouter un ou des mutex non précisé(s) dans l'énoncé !

Autres bonus possibles

- Donnez la possibilité aux 2 joueurs de jouer sur des claviers différents. Pour cela, le threadClavier devrait se mettre en attente de réception sur une file de messages. Un processus indépendant par joueur pourrait alors, dès lecture de la touche enfoncée par ReadChar(), envoyer un message au thread clavier, message qui contiendrait le numéro du joueur ainsi que le code de la touche enfoncée. On pourrait dès lors imaginer qu'il y ait plus que 2 joueurs dans la même partie ☺.
- Un fichier de scores serait agréable.

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)/Thread2011**, 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 !