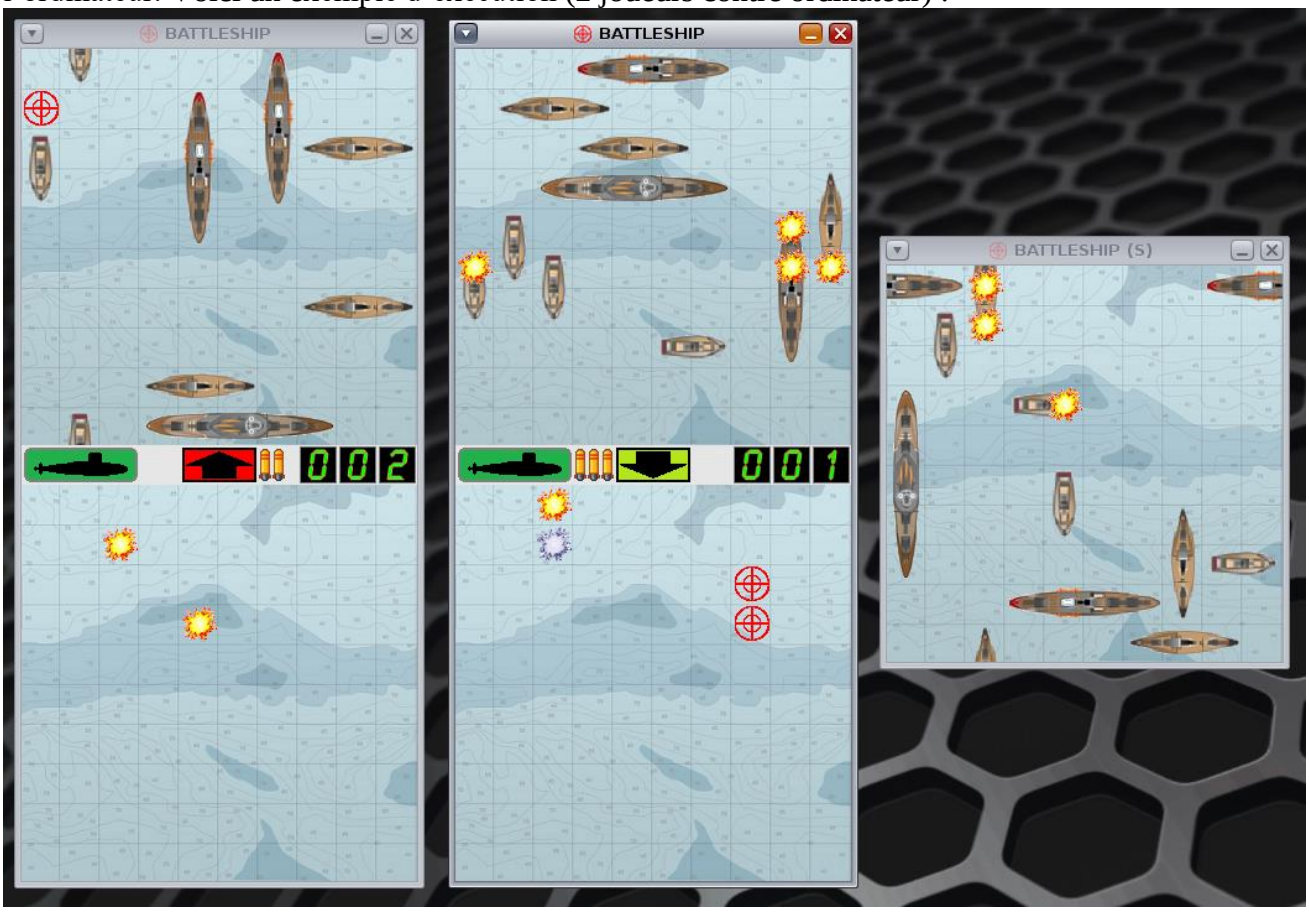
# Laboratoire de Threads - Enoncé du dossier final
# Année académique 2016-2017
# Jeu de combat naval « BattleShip »

Il s'agit de créer un jeu du type « combat naval » multi-joueurs jouant tous contre l'ordinateur (et non l'un contre l'autre). Une autre originalité par rapport à la version classique est que les bateaux se déplacent en permanence ; enfin jusqu'au moment où ils sont touchés. Un bateau touché s'immobilise et une fois toutes ses cases touchées, il coule et disparaît de la grille de jeu. L'ordinateur est modélisé par un processus « Serveur » qui ne meurt jamais. L'ordinateur dispose de 10 bateaux et dès qu'un bateau est coulé, un nouveau apparait. Tous les joueurs disposent de leur propre processus « BattleShip » et jouent contre le même « ordinateur », ils tirent donc tous dans la même « mer » et peuvent donc toucher les mêmes bateaux. Notons dès à présent que deux joueurs ne peuvent tirer au même endroit en même temps. Un système de verrouillage de cible (exclusion mutuelle) sera donc mis en place. Chaque joueur dispose d'un écran double. En haut, il voit ses bateaux se déplacer et se faire toucher/couler par l'ordinateur. En bas, il s'agit de la grille dans laquelle il tire. Bien sûr, il ne voit pas les bateaux de l'ordinateur. A chaque tour, il peut tirer 3 fois. A ce moment, c'est au tour de l'ordinateur de tirer 3 fois. Et ainsi de suite. A chaque fois que le joueur touche un bateau de l'ordinateur, il gagne un point. Le score s'affiche à droite entre les deux grilles. Le joueur dispose également d'un bouton « sous-marin ». Lorsqu'il clique dessus, tout se passe comme s'il envoyait un sous-marin espion en reconnaissance afin de détecter la position des bateaux de l'ordinateur. Un des bateaux de l'ordinateur apparait alors dans la grille du bas pendant un temps déterminé. Le fait de lancer un sous-marin coûte au joueur un tir (sur les 3 dont il a droit par tour) et un point de son score. Le joueur termine sa partie lorsque tous ses bateaux ont été coulés par l'ordinateur. Voici un exemple d'exécution (2 joueurs contre ordinateur) :



Notez dès à présent que plusieurs librairies vous sont fournies dans le répertoire **/export/home/public/wagner/EnonceThread2017**. 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, morceau de bateau, cible, explosion, 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 (cible, explosion, …) dans la fenêtre graphique.
- **MessageQueue** : une librairie C++ encapsulant et simplifiant la gestion d'une file de messages. Celle-ci permettra de gérer la communication entre les processus BattleShip et le Serveur. Des explications sont fournies dans le fichier MessageQueue.h

De plus, vous trouverez les fichiers **BattleShip.cpp** et **Serveur.cpp** qui contiennent déjà les bases de votre application (ouverture de la fenêtre de jeu, connexion par file de messages) et dans lequel vous verrez des exemples d'utilisation de la librairie GrilleSDL, du module Ressources et de la librairie MessageQueue. Vous ne devez donc en aucune façon programmer la moindre fonction qui a un lien avec la fenêtre graphique ou la file de messages. 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… Que ce soit au niveau d'un joueur ou de l'ordinateur, la « mer » dans laquelle se déplacent des bateaux est modélisée par un tableau à 2 dimensions (variable **tab**), de taille 10x10, défini en global. Ce tableau contient des entiers dont la valeur correspond à

- 0 : case non occupée par un bateau
- Une valeur entière positive correspondant au **tid** (« thread id » ou pthread_t) du thread Bateau (**non touché**) qui occupe cette case (voir plus bas).
- Une valeur entière négative correspondant à **-tid**, où tid est l'identifiant du thread Bateau **touché** 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 morceau de Destroyer 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
DessineBateau(7,3,DESTROYER,…); // fonction du module Ressources,
                                 // pour dessiner 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 : Création du thread Bateau (Serveur.cpp)

Attaquons-nous tout d'abord aux bateaux du processus Serveur. Dans un premier temps, le thread principal ne lancera qu'un seul threadBateau. Un bateau occupe plusieurs cases dans la grille de jeu et est caractérisé par la structure Bateau (définie dans le fichier protocole.h). Cette structure contient

- Un **type** (entier) pouvant prendre une des valeurs CUIRASSE (5), CROISEUR (4), DESTROYER (3), TORPILLEUR (2). Ces 4 macros sont définies dans le fichier Ressources.h. Remarquez que ces valeurs de macro correspondent également à la longueur des bateaux. Exemple : Un croiseur occupe 4 cases consécutives dans la grille.
- Des coordonnées **L** et **C** : position (ligne et colonne) de la case du bateau située le plus à gauche et le plus haut dans la grille.
- Une **direction** (entier) pouvant prendre une des valeurs HORIZONTAL ou VERTICAL (macros définies dans Ressources.h). En effet, un bateau peut donc être (et se déplacer) vertical(ement) ou horizontal(ement).
- Un **sens** (entier) pouvant prendre une des valeurs DROITE, GAUCHE, HAUT, BAS (macros définies dans protocole.h).

Cette structure devra être allouée dynamiquement par le thread principal. Une fois allouée, le thread principal initialise uniquement le type et la direction du bateau (c'est le thread bateau lui-même qui gèrera sa position (L,C) et son sens de déplacement). Le thread principal lance alors le thread bateau avec la structure Bateau passée en paramètre.

Une fois lancé, le thread Bateau

- doit tout d'abord déterminer sa position initiale dans la grille tab. Celle-ci sera choisie de manière aléatoire de telle sorte que le bateau ne se positionne que sur des cases libres. **Attention !!!** On impose également qu'il ne puisse y avoir qu'un seul bateau horizontal par ligne (au maximum) et qu'un seul bateau vertical par colonne (au maximum). Pour cela, vous devez mettre en place et utiliser 2 variables globales **lignes[10]** et **colonnes[10]** (de type int ou char) contenant des 1 aux indices des lignes (colonnes) déjà occupées (0 sinon). Dès que toutes les conditions sont réunies, le thread Bateau place son tid (utilisation de **pthread_self()**) dans tab et dessine le bateau dans la fenêtre graphique (utilisation de DessineBateau()).
- doit choisir au hasard un sens de déplacement (GAUCHE ou DROITE si direction est HORIZONTAL, HAUT ou BAS si direction est VERTICAL).

Une fois positionné dans la grille, le thread bateau va gérer seul son déplacement :

- Il se déplace toutes les « delai » nanosecondes d'une case selon son sens, où delai est une variable de type struct timespec contenant une durée aléatoire comprise entre 1.000.000.000 et 3.999.999.999 nanosecondes (utilisation de **nanosleep**). Tous les bateaux ne se déplacent donc pas à la même vitesse.
- S'il atteint le bord de la grille, il continue tout de même son déplacement dans le même sens, son extrémité se retrouvant simplement « de l'autre côté » de la grille. Il s'agit donc d'un déplacement « circulaire » sur une même ligne/colonne.
- Les bateaux ne peuvent pas se chevaucher. Lorsqu'un bateau rencontre un autre bateau, au hasard, soit il ne fait rien (2 chances sur 3), soit il inverse son sens de déplacement (1 chance sur 3).

Tous les threads bateau utilisent les mêmes variables globales : tab[10][10], lignes[10] et colonnes[10]. Afin d'assurer l'accès mutuellement exclusif à ces variables, vous devrez utiliser le mutex **mutexMer**.

Afin de vérifier et valider la mise au point du thread Bateau et du mutex, vous pouvez (à partir du thread principal) lancer un maximum de bateau (20 au plus sinon certains bateaux ne trouveront jamais de place pour entrer…) et réduire le delai (pour accélérer le déplacement des bateaux). N'hésitez pas à tester sans et avec le mutex afin de vérifier son utilité !

## Etape 2 : Création du thread Amiral (Serveur.cpp)

Au final, ce ne sera pas le thread principal qui lancera les threads bateau. Ce sera le rôle du **thread Amiral** de créer les threads bateau et de maintenir le nombre de bateaux constant en permanence. Il devra donc être réveillé à chaque fois qu'un bateau est coulé.

A partir du thread principal, lancer le thread Amiral. Une fois lancé, celui-ci entre dans une boucle infinie dans laquelle il se met en attente de la réalisation de la condition (à l'aide d'un **mutex mutexBateaux** et d'une **variable de condition condBateaux**) suivante :

**« Tant que (nbBateaux >= NB_BATEAUX), j'attends… »**

où

- **nbBateaux** est une variable globale représentant le nombre de bateaux présents dans la grille de jeu. Cette variable doit être en permanence égale à la somme de 4 autres variables globales **nbCuirasses**, **nbCroiseurs**, **nbDestroyers** et **nbTorpilleurs** représentant le nombre de bateaux de chaque type présents réellement dans la grille de jeu.
- **NB_BATEAUX** est une macro égale à la somme de 4 autres macros **NB_CUIRASSES**, **NB_CROISEURS**, **NB_DESTROYERS** et **NB_TORPILLEURS** représentant chacune le nombre de bateaux de chaque type que l'on souhaite. Ces valeurs sont respectivement 1 cuirassé, 2 croiseurs, 3 destroyers et 4 torpilleurs. Dès lors, NB_BATEAUX est égal à 10.

Une fois réveillé (bien sûr, si la condition n'est pas vraie, le thread n'a pas besoin d'être réveillé), le thread Amiral doit

- allouer dynamiquement une structure Bateau dont il doit initialiser le type et la direction de telle sorte que le nombre de bateaux de chaque type voulus soient respectés et qu'il y ait, dans la mesure du possible, autant de bateaux horizontaux que verticaux. Pour cela, deux autres variables globales **nbVerticaux** et **nbHorizontaux** seront utilisées. Toutes les variables globales nbXXX seront protégées par le même mutexBateaux.
- incrémenter les variables nbXXX correspondantes.
- lancer le thread Bateau avec la structure Bateau passée en paramètre.
- remonter dans sa boucle et se remettre en attente (ou non) sur la condition.

Les variables globales nbXXX seront décrémentées par les threads Bateau lorsqu'ils se termineront (bateau coulé). A chaque décrémentation, le thread Amiral sera réveillé par un **pthread_cond_signal** (voir plus bas).

## Etape 3 : Mise en place du serveur de requêtes (Serveur.cpp)

Après avoir lancé le thread Amiral, le thread principal entre dans une boucle infinie dans laquelle

- Il attend une requête provenant des clients (les processus BattleShip). Pour cela, il attend un message de type 1 → requete = connexion.ReceiveData(1) ;
- Crée un **thread Requete** en lui passant en paramètre la requête reçue. C'est le thread Requête qui traitera la requête et répondra au client (la plupart du temps).
- Remonte dans sa boucle et se met en attente d'une nouvelle requête.

Pour information, cette architecture correspond au modèle « Producteur-Consommateur à la demande ». Un thread est créé à chaque requête reçue.

### Une première requête (BattleShip.cpp et Serveur.cpp)

Au démarrage de l'application, après s'être connecté à la file de messages, le processus BattleShip envoie une requête de type « CONNECT » au serveur (tous les types de requête se trouvent dans le fichier protocole.h). Cette requête contient le pid du processus BattleShip.

A la réception de la requête, nous avons vu que le thread principal du Serveur crée un thread Requete qui va, dans le cas d'un CONNECT,

- Mémoriser le pid du client qui se connecte dans un tableau global **pid_t joueurs[10]**. Ce tableau sera utilisé par plusieurs threads, il est donc nécessaire de le protéger par un **mutexJoueurs**.
- Envoyer le pid du serveur au processus BattleShip

La connexion est à présent établie entre le processus BattleShip et Serveur. Le processus BattleShip peut à présent ouvrir sa fenêtre graphique.

## Etape 4 : Création du thread Event (BattleShipp.cpp) et une seconde requête

Afin de permettre au joueur d'interagir avec le processus BattleShip, le thread principal va lancer le **thread Event** 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 **thread Event** va se mettre en attente d'un événement provenant de la fenêtre graphique. Pour récupérer un de ces événements, le thread Event utilise la fonction **ReadEvent()** de la librairie GrilleSDL. Ces événements sont du type « souris », « clavier » ou « croix de la fenêtre ». Les événements du type « clavier » devront être ignorés.

Dans le cas « croix de fenêtre », **le thread Event**

- Envoie une requête du type « DECONNECT » au serveur. Le serveur (ou plutôt le thread Requete créé par le serveur) supprimera simplement le pid du tableau de joueurs connectés.
- Se termine sur un pthread_exit.

Après avoir lancé tous les threads, le thread principal (BattleShip.cpp) se met en attente de la fin du thread Event (utilisation de **pthread_join**), ferme proprement la fenêtre graphique et termine le processus.

## Etape 5 : Les sous-marins et la variable spécifique des bateaux

Dans le processus BattleShip, le bouton « sous-marin » est situé à la ligne 10, colonnes 0, 1 et 2. Lorsqu'il est vert, cela signifie que l'on peut cliquer dessus pour « envoyer » un sous-marin. Lorsqu'il est orange, les sous-marins ne sont pas disponibles (utilisation de DessineBoutonSousMarin()).

Lorsque l'on clique sur le bouton sous-marin, le **thread Event**

- Dessine le bouton sous-marin en orange.
- Envoie un signal SIGUSR1 au Serveur.

**Attention !!!** Seuls les threads Bateau du processus Serveur peuvent recevoir le signal, vous devez donc masquer correctement les signaux dans tous vos threads ! De plus les threads bateau ne peuvent

pas être dérangé n'importe quand. Vous devez donc mettre en place une **section critique** (utilisation de **sigprocmask()**) afin d'éviter que les threads Bateau ne soient dérangés pendant leur déplacement.

Lorsqu'un thread Bateau reçoit le signal SIGUSR1, il doit connaître le pid du processus qui lui a envoyé afin de lui répondre. Pour cela, l'armement de SIGUSR1 doit se faire de manière un peu différente :

```
struct sigaction sigAct;
sigAct.sa_sigaction = HandlerSIGUSR1;
sigAct.sa_flags = SA_SIGINFO;
sigemptyset(&sigAct.sa_mask);
sigaction(SIGUSR1, &sigAct, NULL);
```

et dans le handler :

```
void HandlerSIGUSR1(int sig, siginfo_t *info) {
    printf("pid emetteur : %d",info->si_pid);
}
```

De plus, lorsqu'un bateau reçoit le signal SIGUSR1, on ne sait pas à quel bateau on a affaire et celui-ci n'a pas accès à ses données propres (sa structure Bateau). On vous demande donc de mettre en place une **variable spécifique** du type Bateau pour chaque thread Bateau. En résumé, chaque thread Bateau met l'adresse de sa structure Bateau (reçue en paramètre au démarrage du thread) en zone spécifique (utilisation de **pthread_setspecific** et de **pthread_getspecific**).

Dès lors, dès réception d'un SIGUSR1, le thread Bateau entre dans un handler dans lequel :

- Il récupère sa variable spécifique (adresse de sa structure Bateau)
- Il récupère le pid du processus BattleShip qui lui a envoyé le signal
- Il envoie un message (requête « SOUSMARIN ») au processus BattleShip, ce message contenant sa structure Bateau. (Remarque : le type du message est simplement le pid de BattleShip)

## Création du thread de Réception (BattleShip.cpp)

A partir du thread principal (BattleShip.cpp), lancer le **thread Reception**. La tâche de ce thread est de lire tous les messages provenant du serveur. Il tourne dans une boucle infinie dans laquelle il attend un message dont le type est égal au pid du processus BattleShip → message = connexion.ReceiveData(getpid()) ;

Lorsque le thread Reception reçoit un message « SOUSMARIN », il connaît à présent la position d'un bateau de l'ordinateur. Mais ce n'est pas lui qui va dessiner le bateau, il crée un thread « Affiche Bateau » en lui passant en paramètre la structure Bateau reçue. Cela lui permet de se remettre en attente d'un nouveau message provenant su serveur.

## Création d'un thread Affiche Bateau (BattleShip.cpp)

Une fois démarré la thread Affiche Bateau

- Attend une seconde (utilisation de **nanosleep**)
- Affiche le bateau dans la grille du bas
- Attend 4 secondes
- Efface le bateau de la grille du bas
- Attend 30 secondes avant de remettre le bouton sous-marin au vert et de le rendre à nouveau disponible

- Se termine

## Etape 6 : Gestion des tirs et verrouillage de la cible

Dans le processus BattleShip, la grille du bas permet au joueur de :

- tirer dans la mer de l'ordinateur afin de toucher/couler ses bateaux (apparition d'une cible rouge le temps que le tir aboutisse)
- voir les cases correspondant aux bateaux touchés (explosion orange ou bleue, voir plus loin)
- voir apparaître les résultats de ses sous-marins espions (voir étape 5)
- voir apparaître les bateaux coulés par d'autres joueurs (voir plus loin).

Pour éviter les accès concurrents à cette grille, elle doit également être gérée par un tableau global (dans BattleShip.cpp) **tabTir[10][10]** protégé par un **mutexTabTir**. Ainsi, il sera impossible au joueur de tirer sur un morceau de bateau déjà touché ou sur un bateau affiché suite à l'envoi d'un sous-marin.

### Envoi de la requête

Lorsque l'on clique sur une case vide de la grille du bas, le thread Event dessine une cible rouge à l'endroit du clic (utilisation de DessineCible), flagge **tabTir** en cette case et envoie une requête TIR au serveur. Cette requête contient une structure RequeteTir (définie dans protocole.h) contenant les coordonnées (L,C) de la cible. **Attention !!!** Le thread Event ne doit pas attendre la réponse du serveur. C'est le thread Reception qui gère la réception de tous les messages.

### Réception et traitement de la réponse

En réponse à un TIR, le thread Reception (BattleShip.cpp) va recevoir un message (requête TIR) contenant une structure **ReponseTir** (définie dans protocole.h). Voici cette structure :

```
struct ReponseTir {
  int L ;
  int C ;
  int status ;
  Bateau bateau ;
} ;
```

Evidemment (L,C) correspond aux coordonnées de la cible. Le champ **status** contient un code entier (macros définies dans protocole.h) représentant le résultat du tir :

- **PLOUF** : aucun bateau touché. Dans ce cas, le threadReception efface simplement la cible rouge et libère la case dans **tabTir**.
- **TOUCHE** : un bateau ennemi vient d'être touché. Dans ce cas, le threadReception efface la cible rouge et la remplace par une explosion orange (utilisation de DessineExplosion).
- **DEJA_TOUCHE** : un bateau ennemi est déjà touché par un autre joueur en cette case. Dans ce cas, le threadReception efface la cible rouge et la remplace par une explosion bleue.
- **LOCKED** : un autre joueur a tiré avant vous et donc la cible est déjà réservée (elle est « verrouillée », cela ne veut pas dire qu'un bateau est touché, cela veut dire qu'un autre joueur a tiré avant vous). Dans ce cas, le threadReception efface la cible rouge et la remplace par une cible verrouillée (utilisation de DessineCibleVerrouillee) pendant ½ seconde avant de l'effacer et de libérer **tabTir**.
- **COULE** : un bateau ennemi a été coulé. Les informations complètes de ce bateau se trouvent dans le champ bateau de la structure RequeteTir. Dans ce cas, le threadReception lance un

thread « Affiche Bateau Coule » avec en paramètre la structure bateau reçue. Ce thread affiche le bateau (avec les explosions oranges par-dessus) pendant 3 secondes avant de l'effacer, de libérer **tabTir** et de se terminer.

## Gestion de la requête TIR par le serveur (Serveur.cpp)

Comme pour une requête CONNECT ou DECONNECT (voir plus haut), c'est le thread principal du serveur qui reçoit la requête TIR et la transfère à un thread Requete qu'il crée pour l'occasion.

Le verrouillage de la cible va se faire à l'aide d'un tableau de mutex **mutexCible[10][10]**, chaque mutex de ce tableau permettant de verrouiller une case seule (inutile de « bloquer » toutes les cases alors qu'un tir correspond à une seule case !). Lorsque le mutexCible[i][j] est pris (ou « locked »), cela signifie que la cible (i,j) est verrouillée, sinon elle est libre.

Dans le cas d'une requête TIR, le thread Requete va tout d'abord tenter de verrouiller la cible (utilisation de **pthread_mutex_trylock**). S'il n'y arrive pas (la cible est déjà verrouillée), il envoie directement une réponse de status LOCKED au processus BattleShip et se termine.

Si le thread Requete a réussi à « prendre » le mutex, la cible est verrouillée pour le joueur. Le thread Requete fait alors une **pause de 5 secondes** pour simuler le temps de parcours du missile (utilisation de nanosleep). Trois situations peuvent alors se produire :

- **Soit tab est égal à 0 à l'endroit du tir** : le missile tombe à l'eau et le thread Requete envoie une réponse de status PLOUF au processus BattleShip avant de se terminer.
- **Soit tab est > 0 à l'endroit du tir** : le missile touche un bateau de l'ordinateur et la valeur de tab correspond au tid du thread Bateau touché. Le thread Requete « signale » alors au thread Bateau qu'il a été touché (voir plus bas), met la case de tab à une valeur égale négative à (- pid du processus BattleShip) et se termine. C'est le thread Bateau touché qui répondra au joueur connecté.
- **Soit tab est < 0 à l'endroit du tir** : le bateau a déjà été touché auparavant. Le thread Requete envoie une réponse de status DEJA_TOUCHE au processus BattleShip et se termine.

Remarque : n'oubliez pas de verrouiller le mutexTab avant de tester tab (après la pause de 5 secondes), sinon les bateaux risquent de continuer à bouger pendant votre tir...

## Signalisation et réaction du thread Bateau

Pour signaler à un thread Bateau qu'il a été touché, le thread Requete va lui envoyer un signal SIGUSR2 (utilisation de **pthread_kill**). Dès réception de ce signal, **le thread Bateau entre dans un handler dont il ne sortira qu'à sa mort**.

Le thread Bateau doit connaître la case où il a été touché mais également le pid du processus BattleShip auquel il devra répondre. Pour cela on va utiliser la structure

```
struct comBateau {
  pthread_t tidBateau ;// Identifiant du bateau qui utilise cette structure
  Message Requete[5] ; // Requetes transmises par les threads Requete
  int indEcriture ;    // indice où doit écrire un Thread Requete
  int indLecture ;     // indice où doit lire le thread Bateau
  pthread_mutex_t mutex ; // protège Requete[5], indEcriture, indLecture
  pthread_cond_t cond ;   // synchronisation
} ;
```

Chaque bateau doit disposer en permanence d'une de ces structures. Dès lors, vous devez mettre en place la variable globale

**ComBateau comBateau[NB_BATEAUX] ;**

Au démarrage (juste après sa création), un thread Bateau réservera une de ces structures en affectant le champ tidBateau avec son propre tid (utilisation de pthread_self). Le bateau devra accéder à cette variable. Le plus simple est donc de **mettre l'adresse de cette variable dans une seconde variable spécifique**.

A chaque fois qu'un thread Requete touche un bateau (dont il connaît le tid), il

- Cherche sa structure ComBateau dans la variable comBateau[NB_BATEAUX],
- Ecrit le message reçu de BattleShip à l'indice indEcriture du vecteur Requete[], avant d'incrémenter indEcriture de 1,
- Réveille le thread Bateau par un **pthread_cond_signal** sur la variable de condition de la structure ComBateau.

Dès lors, une fois entré dans son handler de SIGUSR2, le thread bateau

- Récupère sa variable spécifique Bateau afin de savoir qui il est ;
- Récupère sa variable spécifique ComBateau afin de pouvoir communiquer avec les threads Requete ;
- Se met en attente, grâce au mutex et à la variable de condition de sa structure ComBateau, sur la condition

**« Tant que (indLecture == indEcriture), j'attends… »**

- Un fois réveillé, il va lire le message reçu à l'indice indLecture et incrémente indLecture de 1.
- Dessine une explosion orange sur la case touchée.
- Si le bateau est simplement touché, il envoie un message de status TOUCHE au processus BattleShip et se remet en attente sur sa variable de condition
- Si le bateau est coulé, il envoie un message de status COULE au processus BattleShip, ce message contenant sa structure Bateau complète. Ensuite,
    - Il prévient tous les joueurs connectés (sauf celui qui a coulé le bateau) qu'il est coulé. Pour cela, il envoie à chaque processus BattleShip connecté un message BATEAU_COULE contenant sa structure Bateau complète. Dans ce cas, le thread Reception du processus BattleShip lance simplement un thread « Affiche Bateau Coule » (thread déjà décrit plus haut).
    - Fait une pause de 3 secondes (utilisation de nanosleep)
    - Libère les tableaux tab, lignes ou colonnes, et sa structure ComBateau en remettant tidBateau, indEcriture et indLecture à 0.
    - Décrémente nbBateaux de 1 mais également nbXXX (Destroyers, Torpilleurs, Horizontaux, Verticaux , …) de 1 et réveille le thread Amiral par un **pthread_cond_signal** (afin que celui-ci crée un nouveau bateau).
    - Se termine après avoir désalloué sa structure bateau.

## Etape 7 : Création du thread Score (BattleShip.cpp)

La partie d'un joueur se termine lorsque tous ses bateaux ont été coulés par l'ordinateur (voir plus loin). Des plus, l'ordinateur ne perd jamais vu que ses bateaux sont recréés au fur et à mesure qu'ils coulent. Dès lors, le but du jeu pour un joueur est d'accumuler un maximum de points avant que tous ses bateaux ne soient coulés. Donc, un joueur

- Gagne 1 point lorsqu'il touche un bateau (c'est-à-dire lorsque le thread Reception reçoit un message TOUCHE)

- Gagne 2 points lorsqu'il coule un bateau (c'est-à-dire lorsque le thread Reception reçoit un message COULE)
- Perd 1 point lorsqu'il clique sur le bouton « sous-marin ». Le score ne peut jamais être négatif. Dès lors, un joueur ne peut utiliser ses sous-marins que s'il a au moins un point.

Le score du joueur est représenté par une variable globale (de type int) **score**.

A partir du thread principal, lancer le **threadScore** dont le rôle est d'afficher le score à droite entre les deux grilles.

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

**« Tant que ( !MAJScore), 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 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** dans les cases (10,7), (10,8) et (10,9) de la fenêtre graphique (utilisation de **DessineChiffre()**). Ensuite, il doit remettre **MAJScore** à 0 avant de remonter dans sa boucle.

Ce sont les threads Reception et Event qui modifieront le score et réveilleront le threadScore (utilisation de **pthread_cond_signal**).

## Etape 8 : Création des threads Bateau et du thread Amiral (BattleShip.cpp)

Il est temps à présent de s'attaquer aux bateaux du joueur. Leur gestion est tout à fait similaire à celle présente dans le serveur.

A partir du thread principal (BattleShip.cpp), lancez le thread Amiral dont le rôle est de :

- Lancer les 10 threads Bateau du joueur (mêmes types, mêmes nombres de chaque type que dans le serveur, ainsi que la contrainte « un bateau par ligne/colonne »).
- Se mettre en attente sur la condition **(mutex et variable de condition !)**

  **« Tant que nbBateaux > 0, j'attends… »**
- Se terminer dès que le nombre de bateaux est égal à 0, après avoir affiché un « GAME OVER » dans le titre de la fenêtre de jeu.

Les threads Bateau sont quasi identiques (déplacement, réception SIGUSR2, variables spécifiques, structures ComBateau) aux threads Bateau du Serveur aux différences près (dans le handler de SIGUSR2) :

- Ils ne doivent prévenir personne qu'ils se termine (pas d'envoi de message sur la file de message)
- Ils se contente de décrémenter la variable nbBateaux avant de réveiller le thread Amiral (utilisation de pthread_cond_signal) et de se terminer.

Reste maintenant à savoir qui va communiquer avec les threads Bateaux afin de leur faire savoir qu'ils ont été touchés. Il s'agit du thread IA (de BattleShip.cpp) décrit ci-dessous.

## Etape 9 : Création du thread IA (BattleShip.cpp)

A partir du thread principal, lancez le thread IA dont le rôle est de tirer sur les bateaux du joueur. Vous êtes libre d'implémenter la logique de tir que vous voulez dans ce thread. Le plus simple (mais le moins réaliste ☺ !) est de le faire tirer au hasard dans toute la grille de jeu sans tenir compte qu'un bateau a déjà été touché ou pas. Bien sûr, dans ces conditions le jeu peut durer une éternité, dès lors, un système de « nombre de tentatives maximum » va être mis en place.

Une fois démarré, le thread IA tourne dans une boucle infinie dans laquelle il :

- Attend 4 secondes (un tir toutes les 4 secondes)
- Bloque la grille de déplacement des bateaux (mutexTab) pour empêcher que ceux-ci ne se déplacent pendant sa tentative de tir.
- Choisit une case (i,j) au hasard dans tab (ou selon toute autre tactique plus ou moins intelligente ☺) et dessine une cible rouge sur cette case pour montrer au joueur où il tire. Il pourrait au moins éviter de tirer sur une case correspondant à un bateau déjà touché…
- Attend une seconde avant de voir s'il a touché ou pas un bateau.
- Si tab[i][j] = 0, aucun bateau touché. Dans ce cas, il efface la cible rouge, libère la grille de déplacement des bateaux et remonte dans sa boucle.
- Si tab[i][j] > 0, il a touché un bateau dont il connaît à présent le tid. Il peut transmettre au thread Bateau correspondant un « Message » contenant les coordonnées du tir (utilisation de sa structure ComBateau) après lui avoir envoyé un signal SIGUSR2. Il met alors tab[i][j] à une valeur négative avant de débloquer le mutexTab et de remonter dans sa boucle.

Afin que le jeu ne dure pas une éternité, si le thread IA n'a rien touché au bout de 20 tentatives, au lieu de tirer au hasard, il cherche une case positive de tab.

## Etape 10 : Synchronisation du jeu et fin de la partie (BattleShip.cpp)

Le thread IA ne peut tirer toutes les 4 secondes sans s'arrêter, il doit laisser le tour au joueur. De même pour le joueur, il ne peut cliquer sur rien pendant que l'IA joue…

Vous devez donc mettre en place un système permettant la séquence suivante :

- Le joueur tire 3 fois. A chaque tir, un petit missile supplémentaire (utilisation de DessineMissiles) apparaît à gauche de la flèche « verte » (utilisation de DessineFleche). Pendant ce temps, le thread IA doit être bloqué (une **variable de condition** et un **mutex** dédiés seraient les bienvenus ☺).
- Le thread IA tire 3 fois. A chaque tir, un petit missile supplémentaire apparaît à droite de la flèche « rouge ». Pendant ce temps, il doit être impossible au joueur de cliquer sur quoi que ce soit, mis à part la croix de la fenêtre.

Et ainsi de suite, jusqu'au moment où le thread IA aura coulé tous les bateaux du joueur. A ce moment-là, il doit se terminer. De plus, le joueur doit être dans l'impossibilité de cliquer sur quoi que ce soit, mis à part la croix de la fenêtre afin de terminer le thread Event, et l'application elle-même (voir étape 4).

**Remarque** : Une fois que le tout est mis au point, ajouter un argument sur la ligne de commande du serveur afin qu'il puisse être lancé avec ou sans fenêtre graphique (afin de ne pas voir ses bateaux ☺).

## 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 !

## 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)/Thread2017**, 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 !