FROGER Adrien L3 SPI - groupe 2

SE & PS
Rapport TP2

TP 2 **Bataille navale par signaux**

SOMMAIRE

Introduction :

Le but de ce TP était de modéliser un jeu de bataille navale entre plusieurs bateaux qui sont représentés chacun par un processus distinct. Ces bateaux se déplaçant dans un unique fichier mer (ressource critique), leurs actions passent alors par un processus amiral permettant d'éviter tout conflit d'accès dans ce fichier. La communication entre un navire et l'amiral se fait alors par le biais d'un échange de signaux.

Ce rapport permet de comprendre le mode de communication adopté entre l'amiral et les navires, d'aborder les différents signaux utilisés et leur rôle ainsi que les handlers qu'ils déclenchent et la manière dont ces derniers sont appliqués aux signaux.

1. Liste des signaux employés

Voici la liste des signaux employés et leur rôle :

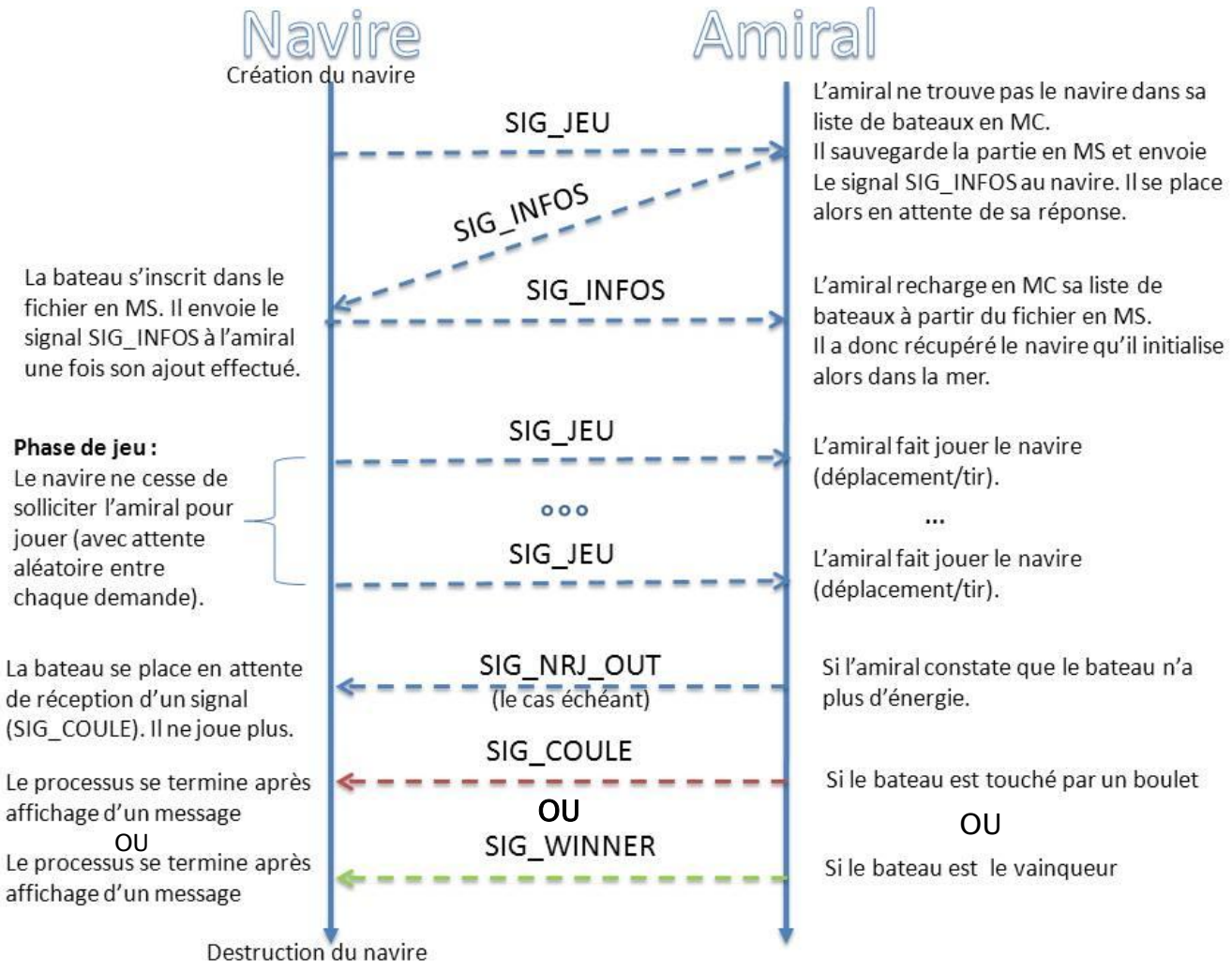
Signal	Rôle de Navire vers Amiral	Rôle de Amiral vers Navire
SIG_JEU	Effectue une demande de jeu auprès de l'amiral.	
SIG_INFOS	Confirme l'écriture des informations du bateau dans le fichier en MS.	Demande l'écriture des informations du bateau dans le fichier en MS en vue de son ajout à la liste en MC.
SIG_NRJ_OUT		Signale au bateau qu'il n'a plus d'énergie
SIG_COULE		Signale au bateau qu'il a été coulé
SIG_WINNER		Signale au bateau qu'il est le vainqueur

La correspondance entre nom de signal personnalisé et nom de signal réel est effectuée dans le fichier signaux.h commun à l'amiral et aux bateaux.

2. Principe de communication retenu

Le principe de communication pour lequel j'ai opté est le suivant : les bateaux n'envoient qu'un seul signal (SIG_JEU) pour s'initialiser/jouer à l'amiral qui gère alors le fait que le bateau en question soit ou non initialisé dans la mer. Pour cela, on se base sur une liste en MC qui contient les bateaux de la partie avec leur statut (actif/inactif). Quand un bateau est actif, il ne cesse envoyer un signal (SIG_JEU) pour demander à jouer. A réception de ce signal, l'amiral va vérifier que le bateau en question est bien présent dans sa liste de bateaux en MC. Si ce n'est pas le cas, il sauvegarde alors cette liste en MS (pour ne pas perdre les informations concernant les autres bateaux), envoie un signal (SIG_INFOS) pour demander au bateau de s'ajouter à ce fichier en MS et attend la réception de ce même signal par le bateau en confirmation que ce dernier s'est ajouter en MS. Une fois le signal de confirmation reçu, l'amiral tente alors d'initialiser le bateau dans la mer. S'il y parvient, le bateau va alors pouvoir jouer, son statut est alors à actif. Si ce n'est pas le cas, le bateau sera à présent en MC avec un statut inactif. Ainsi, lors de sa prochaine demande de jeu, l'amiral pourra tenter de l'initialiser directement sans qu'il n'ait à s'inscrire au préalable dans le fichier en MS.

Un dessin valant mieux qu'un beau discours, voici le chronogramme de ce principe :



Une autre solution aurait été d'enregistrer la partie en MS après chaque tour de jeu et que le bateau s'ajouter directement au fichier en MS dès sa création. L'ajout utilisant un verrou bloquant, cela aurait fonctionné mais cela présente le gros inconvénient de nécessité une quantité importante d'écriture disque.

Comparatif : Si l'on prend 5 bateaux jouant chacun 100 fois, on obtient le comparatif suivant concernant le nombre d'écritures disque :

Solution 1	10 écritures (5 de l'amiral + 5 pour les navires)
Solution 2	505 écritures (5x100 de l'amiral + 5 pour les navires)

L'inconvénient principal du principe pour lequel j'ai opté est le risque d'une attente infinie du côté amiral. En effet, lorsque l'amiral ne connait pas le bateau (bateau non présent en MC), il bloque le jeu (sigsuspend()) et demande au bateau de d'inscrire dans le fichier en MS puis attend confirmation de cette écriture par réception d'un signal SIG_INFO du navire vers l'amiral. Si pour une raison ou pour une autre le navire ne renvoyait pas ce signal, le bateau serait alors infiniment en attente. Afin de palier à cela, on utilise une sorte de timer géré grâce à la fonction alarm() et on définit le masque utilisé pour sigsuspend() de tel sorte qu'il laisse passer SIG_INFOS ET SIGALRM (et pas SIG_JEU car il n'est plus masqué par le mécanisme de sigaction vu que le masque est temporairement remplacé et pourrait interrompre l'attente de manière injustifiée). Ainsi, en cas de défaillance du navire, le signal SIGALRM envoyé par la fonction alarm() nous permettra de reprendre tout de même le jeu.

3. Handlers utilisés

Voici un tour d'horizon des différents handlers utilisés et leur rôle :

3.1 Dans l'amiral

• **hdl_ecriture_ok :** pas de rôle particulier, sert juste à rendre le signal SIG_INFOS capturable (ce qui est nécessaire pour le sigsuspend( ))

• **hdl_sigalrm :** idem pour le signal SIGALRM

• **hdl_jeu :** permet de gérer les demandes de jeu des navires (récupération du navire dans la liste en MC à partir du fichier en MS, initialisation du navire dans la mer, jeu (déplacement/tir), envoi des signaux SIG_NRJ_OUT, SIG_COULE et SIG_WINNER).

3.2 Dans les navires

• **hdl_ecriture_ms :** gère l'écriture des informations du bateau dans le fichier en MS puis envoie le signal SIG_INFO à l'amiral pour confirmer le bon déroulement.

• **hdl_arret_de_jeu :** déclenché par le signal SIG_NRJ_OUT, SIG_COULE ou SIG_WINNER. Ce handler est celui qui est appelé lorsqu'un bateau doit sortir de sa boucle « infinie » de demandes de jeu. Si le signal reçu est SIG_NRJ_OUT, le bateau est alors placé en attente du signal SIG_COULE. En effet, il est dans ce cas sans défense donc se fera tirer dessus d'ici peu. Si le signal reçu est SIG_COULE ou SIG_WINNER, alors un message approprié est affiché à l'écran et le processus se termine.

Remarque : Ma solution ne gère pas les ex-aequo. Il pourrait être possible de rajouter cette gestion avec une détection de ce cas de figure dans l'amiral et un signal SIG_EXAEQUO qui permettrait alors d'afficher un message approprié aux navires concernés avant de quitter. Dans les cas d'essais vus durant la démonstration, les ex-aequo ne risquaient pas d'intervenir mais il serait bon de rajouter cette fonctionnalité car ce cas de figure est tout à fait possible.

4. Application des handler aux signaux

L'application du handler **hdl_arret_de_jeu** aux signaux SIG_NRJ_OUT, SIG_COULE et SIG_WINNER est réalisée à l'aide de la fonction signal. L'inconvénient de cette fonction est qu'elle ne réinstalle pas le handler personnalisé à la fin de son exécution et qu'elle n'empêche pas l'interruption du handler par une autre réception du signal durant l'exécution de ce dernier. Dans le cas du handler hdl_arret_de_jeu, ce n'est aucunement gênant. En effet, on ne quitte ce handler que par un exit donc nous n'avons pas à nous soucier du premier problème. D'autre part, les signaux SIG_NRJ_OUT, SIG_COULE et SIG_WINNER ne peuvent être envoyés simultanément par l'amiral au bateau donc nous ne nous soucions pas du risque d'interruption.

Le reste des handlers est appliqué par le biais de la fonction sigaction aux différents signaux voulus. Par défaut, cette fonction présente l'avantage de réinstaller le handler personalisé à la fin de l'exécution du handler et de masquer le signal détourné durant l'exécution du handler.

Dans navire.c, le **handler hdl_ecriture_ms** est appliqué par sigaction. Le but est de supporter une éventuelle erreur de lecture du fichier en MS par l'amiral. Dans un tel cas, l'amiral renverra alors une demande d'inscription des informations du bateau en MS la prochaine fois que le bateau redemandera à jouer. Il faut donc que le handler personnalisé soit maintenu ce qui est le cas par défaut avec sigaction.

Dans amiral.c, on remarque que le champ sa_flags de la structure sigaction qui sera appliquée pour le détournement du signal SIG_JEU comporte l'option **SA_SIGINFO**. Cette option permet de récupérer une structure siginfo contenant notamment le numéro de PID du bateau qui envoie le signal ; donnée dont nous avons besoin. Le champ sa_handler est alors remplacé par l'usage du champ sa_sigaction permettant un prototype de handler incluant 3 arguments et non plus 1 à savoir : le numéro du signal, un pointeur vers la structure contenant les différentes informations concernant le signal et un pointeur de contexte. Le PID de l'expéditeur du signal est alors récupérable dans le handler (champ si_pid de la structure siginfo récupérée).

Enfin, toujours dans amiral.c, on peut remarquer le masquage du signal SIG_INFOS pour le handler de jeu **hdl_jeu**. Ce masquage permet de ne pas être interrompu par un signal SIG_INFOS qui pour une raison ou une autre arriverait de manière tardive d'une précédente exécution d'un handleur de jeu pour un bateau non présent en MC.

5. Validation tardive (les boulets)

Je n'ai pu faire valider mon TP que le lendemain de la date prévue initialiement. Ceci est dû au fait qu'auparavant, lorsqu'un bateau tirait sur un autre, le boulet s'imprégnait dans la coque du bateau victime qui n'était coulé que lors de sa prochaine demande de jeu. Ce mode de fonctionnement ne correspondant pas aux attentes, j'ai alors modifié mon code pour que le coulage du bateau soit effectif dès le tir effectué. Je pensais qu'il suffisait « simplement » de récupérer l'indice du bateau visé par le biais de la fonction *bateau_liste_coord_chercher()* et de couler le bateau qui se trouvait à cet indice dans la liste mais lorsque j'ai mis en place ce mode de fonctionnement, j'ai alors eu des problèmes tels que des bateaux qui « se coupaient en deux ». Parti à priori sur un bug d'affichage ou d'attente, j'ai réalisé un peu tardivement que le problème provenait de la fonction *bateau_liste_coord_chercher()*. En effet, lorsqu'un bateau est coulé, bien que son processus soit tué et qu'il soit effacé de la carte, il est encore dans la liste en MC et donc, lorsque l'on récupère l'indice du bateau victime du tir, cela nous renvoyait l'indice du premier occupant de la case et non nécessairement de l'occupant actuel.

Ne disposant pas de fonction pour supprimer un bateau de la liste, j'ai pallié à ce problème en réalisant une fonction *bateau_rm_coord(bateau_t * bateau)* qui permet d'inscrire -1 à la place des coordonnées du corps du bateau passé en paramètre. Cette fonction est alors invoquée lorsqu'il faut couler un bateau. Ainsi, l'indice d'un bateau coulé ne sera plus susceptible d'être remonté par le biais de la fonction *bateau_liste_coord_chercher()*.

Voici le code de la fonction *bateau_rm_coord(bateau_t * bateau)* :

```
void bateau_rm_coord(bateau_t * bateau)
{
    coord_t co_vide ;
    coord_ecrire( &co_vide , -1 , -1 , -1 );
    int i;
        for( i=0 ; i<BATEAU_TAILLE ; i++)
     coord_affecter( &(bateau->corps[i]) , co_vide );
}
```

Ainsi, on peut couler un bateau dès que ce dernier est touché sans aucun problème. Il n'est alors même plus nécessaire d'utiliser la fonction *mer_bateau_cible _tirer()* qui inscrit le '@' du boulet de canon dans la case visée.

Je fais toujours appel à cette fonction dans mon code mais seulement à des fins de clarté d'affichage : on voit le boulet à l'écran et on comprend donc pourquoi les bateaux disparaissent de la carte mais commenter la ligne *mer_bateau_cible_tirer()* n'impacterais en rien le bon déroulement de la partie.

Conclusion :

Ce TP m'aura permis d'aborder le fonctionnement des signaux à savoir leur détournement pour effectuer un traitement en particulier (handler) à leur réception et les problématiques de masquages.

J'ai pu constater les différences fondamentales entre la fonction sigaction() et la fonction signal() et observer comment récupérer le PID expéditeur d'un signal (ainsi que d'autres informations sur le signal).

Enfin, j'ai également pu comprendre le fonctionnement de la fonction sigsuspend() qui permet de remplacer temporairement le masque des signaux et d'attendre la réception d'un signal non masqué ce qui m'aura été utile à la gestion d'un timer.