INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

SDE – Compte-rendu Projet : Secret Défense

Erwan Etillieux, Pierre Fleitz (3TC-1)

Janvier 2015

Table des matières :

1. Présentation du TP :

Ce TP consiste à simuler le transfert de messages codés, via un réseau, d'un poste émetteur vers un poste récepteur. On précise bien qu'il s'agit ici d'une simulation car en vérité les postes récepteurs et émetteurs seront un seul et même poste qui sera le nôtre.

Nous allons effectuer cette simulation à l'aide de 5 processus qui communiqueront grâce à des IPC : mémoire partagée, tube nommé, signaux, file de messages ou sémaphores.

La finalité de ce TP réside dans le fait que la phrase soit au final décodée et rangée dans un fichier nommé : SecretDefense.

2. Conception :

2.1 Démarrage de l'application :

Nos 5 processus doivent pouvoir être lancés les uns après les autres sans problème.

Le processus SAISIE va créer l'IPC le liant à CODE (tube nommé TUBE), de ce fait ce sera SAISIE qui devra être lancé le premier. CODE qui créé quasiment toutes les autres IPC devra être lancé en second et les 3 autres processus peuvent être lancés dans n'importe quel ordre puisque de toutes façons CODE et CHOIX sont synchronisés par sémaphore, et RECEPTION et DECODE doivent attendre que la file de message contienne des messages.

Pour le lancement ordonné des processus, nous prévoyons d'écrire un script « lancement » adapté à l'utilisation de notre application. Celui-ci permet tout simplement d'ouvrir 5 terminaux, un pour chaque processus lancé et dans l'ordre. En effet les scripts effectuent les actions de façon séquentielle.

2.2 Arrêt de l'application :

Pour faire en sorte que l'application soit propre, elle doit pouvoir se terminer proprement. C'est à dire qu'un processus doit se détacher, voire supprimer les IPC auxquelles il est connecté avant de se couper. Nous prévoyons donc une fonction quitter() dans chaque processus qui s'occupera de cette tâche. Lorsqu'un processus recevra le signal SIGINT, une fonction signal le déroutera vers la fonction quitter().

Par exemple, dans le processus RECEPTION, quitter() permet de détacher le segment de mémoire alloué dans la mémoire partagée, puis de la détruire. De plus RECEPTION étant responsable de la création de deux sémaphores, quitter() permettra de les détruire.

De même que pour le lancement de l'application, nous avons écrit un script « stop » qui envoie un signal SIGINT à tous les processus ouverts dans l'ordre inverse d'ouverture de « lancement ».

2.3 Les IPC :

2.3.1 Les clés des IPC :

Pour que les processus puissent s'attacher aux IPC, ils doivent connaître la clé associée aux IPC en question.

Cependant il est impossible de communiquer via une IPC sans y être attaché et nous n'utilisons aucun père/fils (nous avons en effet décidé de synchroniser nos processus par le biais de sémaphores). Donc, le processus créateur de l'IPC n'a aucun moyen pour divulguer la clé à un autre processus, à moins d'utiliser une IPC déjà existante mais le problème reste le même pour cette dernière. Dans un soucis de simplicité nous avons choisi les clés nous même, et nous les avons déclaré en temps que variable globale dans les processus qui en ont besoin.

Nous avons décidé que les clés pour les sémaphores seraient 101, 102, 103, 104, 105 et 106 (dans l'ordre de création des sémaphores, 101 pour la première et 106 pour la dernière). La clé 200 est définie pour la file de message, et la clé 300 pour la mémoire partagée.

2.3.2 Le tube nommé "TUBE" :

Pour faire transiter ce qu'écrit l'utilisateur de l'application du processus SAISIE vers le processus CODE, nous utilisons un tube nommé. Il n'est donc pas nécessaire que CODE soit le processus fils de SAISIE, qui sera le créateur du tube. L'important est que le tube soit ouvert en lecture dans CODE grâce à l'option O_RDONLY, qui est bloquante si aucun processus ne l'a ouvert en écriture. On ouvre donc le tube en écriture dans SAISIE en utilisant O_WRONLY.

La fonction quitter() de SAISIE s'occupera de détruire le tube.

2.3.3 La file de message :

La file de message (créée par CODE) est reliée à 4 processus. (CODE, CHOIX, RECEPTION, DECODE). Parmi ces 4 processus, CODE et CHOIX doivent y déposer des messages (de types différents), et RECEPTION et DECODE doivent en récupérer.

Comme indiqué dans le sujet, nous devons envoyer 2 types de datagrammes. Un datagramme « donnée » contient le numéro de la phrase, plus la phrase codée. Un datagramme « code » contient le numéro de la phrase, plus la lettre associée au code de cette phrase. Nous avons donc défini une structure msgbuf qui nous permettra d'envoyer deux types de messages.

Notre structure msgbuf possède 4 champs, long mtype pour le type du message, int numP pour le numéro de la phrase codée, int numL pour la valeur du décalage de codage, et enfin un char mtext[64] pour la phrase codée.

Les messages de mtype 1 sont les messages envoyés par CHOIX qui ont le champ mtext NULL, les messages de mtype 2 sont envoyés par CODE, et ont le champ numL NULL.

2.3.4 Les Sémaphores :

Nous avons 6 sémaphores : idS1, idS2, idS3, idS4, idS5 et idS6.

Le sémaphore idS1 créé par CODE, permet d'envoyer le numéro de la phrase à CHOIX. Pour se faire on l'initialise à la valeur de numPhrase, et CHOIX le récupèrera avec la fonction getVal(idS1) implémentée dans sémaphore.h.

De la même façon que idS1, idS2 permet d'envoyer le numLettre (valeur de décalage pour le codage) de CHOIX à CODE. C'est en effet CHOIX (d'où son nom) qui va décider à partir de quelle lettre on codera la phrase.

Les sémaphores idS3 et idS4 sont des sémaphores de synchronisation entre CODE et CHOIX.

Les sémaphores idS5 et idS6 sont des sémaphores de synchronisation entre DECODE et RECEPTION.

2.3.5 La mémoire partagée :

La mémoire partagée est créée par RECEPTION. À l'aide de la fonction attach_shmem implémentée dans shmem.h, nous allouons un segment de mémoire dans la mémoire partagée pour y accueillir la phrase que l'on veut envoyer. On initialise ce segment de mémoire comme une chaîne de caractères nommée buffer et que l'on initialise avec des caractères NULL (0). Cette opération nous permettra plus tard de remplir buffer avec la phrase que l'on veut envoyer à DECODE. Une fois ceci fait nous avions deux possibilités pour remplir buffer de la phrase à décoder, utiliser la fonction strcpy qui prend en paramètres l'adresse destination, et l'adresse source, ou une boucle for dans laquelle on remplirait « manuellement » le buffer. Nous avons tout simplement décidé d'utiliser ces deux techniques. Une première fois nous avons utilisé strcpy pour placer le message issu de la file de message dans une chaîne de caractère préalablement déclarée « tab », que l'on affiche avec un printf (pour vérifier la réception de la bonne phrase), puis une boucle for pour remplir le buffer avec tab.

Pour empêcher à DECODE et RECEPTION d'utiliser le segment de mémoire partagée en même temps nous avons décidé de les synchroniser à l'aide de deux sémaphores (idS5 et idS6).

2.3.6 Les tests :

Pour s'assurer du bon fonctionnement de l'application, il est nécessaire de mettre en place une série de tests.

Nous avons mis des tests qui vérifient la validité de toutes les créations d'IPC, ainsi que la validité des attachements à ces IPC. Cela consiste à vérifier que les appels aux fonctions xxxget ne retournent pas -1. Dans le cas contraire un message d'erreur s'affiche et le processus s'arrête.

De plus, afin de vérifier que les messages sont envoyés et reçus correctement nous avons mis en place aux endroits stratégiques (après la réception de la phrase à coder etc.) des printf qui nous confirment la réception du bon message.

3. Réalisation :

3.1 SAISIE :

Dans une première version de SAISIE nous utilisions la fonction scanf pour récupérer les mots tapés par l'utilisateur. Cependant nous nous sommes vite rendus compte que cette solution ne nous permettait pas d'écrire des phrases mais seulement un mot. En effet scanf ne reconnaît pas le caractère [espace], il s'arrêtera à la fin du premier mot. Pour apporter une solution à ce problème, nous avons décidé d'utiliser la fonction gets. Celle-ci nous permet en effet d'envoyer des phrases entières dans le tube nommé.

C'est dans SAISIE que nous avons décidé de gérer le fait que l'on ne peut pas avoir de phrases de plus de 64 caractères. Pour se faire nous avons utiliser un if : dans le cas ou la taille de la phrase rentrée dépasse 64 caractère, SAISIE renvoie un message d'erreur et demande à l'utilisateur de rentrer une phrase moins longue. Dans le cas où la phrase est de bonne taille on écrit dans le tube nommé avec la fonction write.

3.2 CHOIX :

Le processus choix a un objectif principal qui est de choisir le décalage qu'il y aura entre le caractère tapé et le caractère codé. Nous sommes partis du principe que l'utilisateur tape uniquement des caractères de l'alphabet en minuscule. Le décalage doit donc aller jusqu'à 26. Pour choisir aléatoirement ce décalage, nous avons utilisé la fonction (rand()%26)+1 qui donne une valeur allant de 1 a 26.

3.3 CODE :

Le but du processus CODE est de transformer la phrase tapée au clavier, en une phrase codée n'ayant plus aucun sens quand on la lit. Pour simplifier nous avons décidé que les phrases codées, comme les phrases tapées au clavier, ne contiennent que des minuscules de l'alphabet et des espaces qui eux ne seront pas codés. Pour le codage d'une phrase, le principe est d'ajouter au caractère de base, la valeur du décalage choisi par CHOIX.

Cependant, en faisant cette opération, on peut obtenir des caractères qui ne sont plus dans l'alphabet. En effet, le code ASCII du 'z' est 122, ce qui implique que nous ne devons pas dépasser 122. Ainsi nous avons mis en place un if qui enlève 26 aux valeurs dépassant 122. Comme s'il s'agissait d'une boucle, après 'z' on repart à 'a'. 123-26=97 le code ASCII de « a ». Pour que ce système fonctionne, nous avons pris la précaution de bien caster nos char.

Nous avons en effet longtemps rencontré un problème lié au fait que nos chars n'étaient pas castés. Le code ASCII étant codé de 127 à -127 pour les caractères imprimables lorsque le code de la lettre à coder + le décalage dépassaient 127, la valeur devenait négative, et donc le codage défaillant.

3.4 DECODE :

Pour ce qui est de DECODE, le principe est tout simplement de lire la phrase depuis la mémoire partagée, la valeur du décalage depuis la file de message, puis d'effectuer l'opération inverse du codage. Au lieu d'enlever 26 si nous dépassons 122, cette fois nous rajoutons 26 si nous sommes inférieurs à 97.

DECODE doit ensuite écrire la phrase codée dans un fichier SecretDefense, ce que l'on a fait avec un fprintf de la phrase décodée.

Lors de l'exécution de DECODE nous nous sommes rendus compte que lorsque l'on décodait une phrase moins longue que la précédente, seuls les premiers caractères correspondant à la taille de la nouvelle phrase étaient « écrasés » mais il restait les caractères de l'ancienne phrase. Pour résoudre ce problème nous avons imposé à chaque début de boucle de DECODE de réinitialiser la phrase à décoder à l'aide de la fonction memset.

3.5 RECEPTION :

RECEPTION va simplement lire la valeur du décalage et le numéro de la phrase dans la file de message. Mais son action principale va être de créer la mémoire partagée entre lui et DECODE.

C'est alors ici que l'on a rencontré notre plus gros problème, nous avons eu pendant un long moment un problème de segmentation dû au fait de notre mauvaise gestion de la taille du segment de mémoire partagé alloué (buffer).

4. Préparation de l'Implémentation :

4.1 Les macro-algorithmes :

```
SAISIE :
1.1.
Rediriger SIGINT vers la fonction quitter()
Créer le tube nommé
S'attacher au tube nommé
TOUJOURS RÉPÉTER :
        Saisir phrase à coder au clavier
        Envoyer la phrase dans le tube nommé
FIN TOUJOURS RÉPÉTER

FONCTION quitter
        Se détacher du tube nommé
        Détruire le tube nommé
        Terminer SAISIE
FIN FONCTION quitter
```

CHOIX :

Rediriger SIGINT vers fonction quitter()
S'attacher à la file de message
S'attacher à idS1
S'attacher à idS2
S'attacher à idS3
S'attacher à idS4
TOUJOURS RÉPÉTER
    Récupérer numPhrase dans idS1
    Choisir numLettre
    Initialiser idS2 avec la valeur numLettre
    Envoyer la lettre et le numéro de phrase dans la file de message
FIN TOUJOURS RÉPÉTER

FONCTION quitter
    Terminer CHOIX
FIN FONCTION quitter

CODE :

Rediriger SIGINT vers la fonction quitter()
Créer la file de message
Créer le sémaphore idS1
Créer le sémaphore idS2
Créer le sémaphore idS3
Créer le sémaphore idS4
TOUJOURS RÉPÉTER :
    S'attacher au tube nommé
    Lire le message dans le tube nommé
    Attribue un numéro à la phrase
    Initialise idS1 avec numPhrase
    Récupérer numLettre dans idS2
    Code la phrase
    Sleep(1)
    Affiche la phrase codée
    Envoie la phrase codée dans la file de message
FIN TOUJOURS RÉPÉTER

FONCTION quitter
    Supprimer la file de message
    Supprimer idS1
    Supprimer idS2
    Supprimer idS3
    Supprimer idS4
    Terminer CODE
FIN FONCTION quitter

RECETPION :

Rediriger SIGINT vers fonction quitter()
Créer la mémoire partagée Shmem
Créer le sémaphore idS5
Créer le sémaphore idS6
S'attacher à la file de message
TOUJOURS RÉPÉTER
    Lire la phrase codée dans la file de message
    La stocker dans un buffer
    Down idS6
    Ecrire la phrase dans la mémoire partagée
    Up idS5
FIN TOUJOURS RÉPÉTER

FONCTION quitter
    Se détacher de la mémoire partagée
    Supprimer la mémoire partagée
    Supprimer idS5
    Supprimer idS6
    Terminer RECEPTION
FIN FONCTION quitter

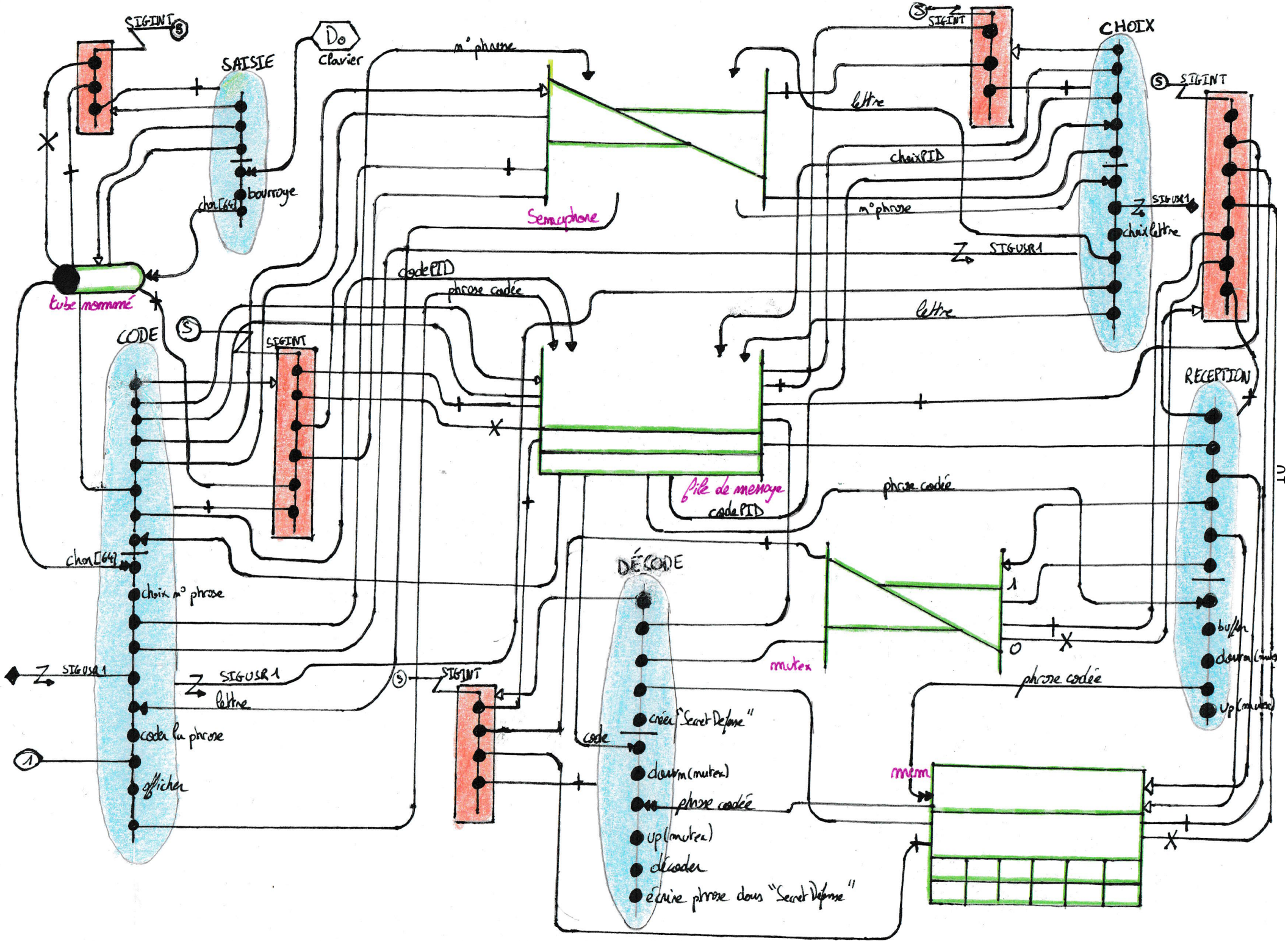
DECODE :

Rediriger SIGINT vers fonction quitter()
S'attacher à la file de message
S'attacher à idS5
S'attacher à la mémoire partagée
Créer le fichier SecretDefense
TOUJOURS RÉPÉTER
    Récupérer le code (numLettre + numPhrase) dans la file de message
    Down (idS5)
    Récupère la phrase codée dans la mémoire partagée
    Up (idS6)
    Décoder la phrase codée
    Afficher la phrase décodée
    Ecrire la phrase décodée dans SecretDefense
FIN TOUJOURS RÉPÉTER

FONCTION quitter
    Terminer DECODE
FIN FONCTION quitter

4.2 Diagramme de Lâcatre :

Il ne faut pas oublier que le diagramme de Lâcatre présent à la page suivante est le diagramme que nous avons dessiné et mis en place pour la préparation de ce Projet. On peut y voir la présence de signaux qui ne sont plus présents dans notre code. En effet, une fois passés à la pratique nous avons dû faire certains choix. Parmi certains d'entres eux nous avons décidé de ne pas passer par une synchronisation père/fils entre CODE et CHOIX mais par une synchronisation avec des sémaphores. Nous voulions éviter au maximum d'avoir des relations père/fils dans notre code. Ce diagramme nous a été très utile tout au long de la réalisation de notre projet.


SIGINT
SAISIE
Do
Clavier
n° phrase
Sémaphore
lettre
SIGINT
CHOIX
SIGINT
choixPID
n° phrase
SIGUSR1
choix lettre
SIGUSR1
tube nommé
codePID
phrase codée
lettre
CODE
SIGINT
RECEPTION
file de message
codePID
phrase codée
chan[64]
choix n° phrase
DÉCODE
1
0
mutex
SIGUSR1
SIGUSR1
lettre
SIGINT
code
crée "Secret Défense"
dors(mutex)
phrase codée
up(mutex)
décoder
écrire phrase dans "Secret Défense"
phrase codée
mem
buffer
dors(mutex)
Up(mutex)

5. Conclusion :

Lors de ce projet, d'un point de vu technique nous avons manipulé 5 outils différents permettant de communiquer entre différents processus. Nous avons pu voir la limite de chacun de ces outils : leur mise en place, leur efficacité, les problèmes de synchronisation liés à leur utilisation. Nous avons pu mettre en pratique tout le contenu du cours ou presque.

D'un point de vu humain, ce projet nous a aussi, et surtout, permis de se plonger pour la première fois dans nos études autour d'un vrai cahier des charges à respecter, de reconsidérer l'importance d'une bonne préparation, d'une bonne stratégie avant même d'avoir commencer à allumer un ordinateur pour coder. En plus de tout ça, ce projet n'a fait que renforcer notre vision sur l'importance du travail d'équipe, du partage de savoir (entre partenaires voire même entre différents groupes), de la patience et de la persévérance. Il apparaît aux yeux de notre équipe comme étant une réussite et un véritable outil pédagogique.