IN405 – Système d'exploitation
TD 1 – Terminal et script Shell

S. Gougeaud

2017/2018

L'archive `code-td1.tar` *contient un fichier :*

- `code-mystere.c` – *fichier principal de l'exercice 5, à déboguer.*

Exercice 1 – Compréhension des commandes de base

Soit la liste de commandes suivante : `cat cd cp diff echo gcc gdb ls make man mkdir mv rm rmdir sudo tar time touch vi`

1. Donnez une brève description pour chacune des commandes.
2. Quelles commandes consistent en l'exécution d'un binaire ?
3. Quels chemins sont représentés par les symboles suivants : `.`, `..`, `~`.

Exercice 2 – Première utilisation du terminal

Pour chacune des questions suivantes, exécutez la commande correspondante.

1. Déplacez vous dans le répertoire temporaire de votre système de fichiers.
2. Créez le répertoire `project` ainsi que les sous-répertoires `doc`, `include` et `src`.
3. Au sein du dossier `project`, créez un fichier `README` contenant votre nom et prénom. Créez le fichier `func.h` dans `include`, les fichiers `main.c` et `func.c` dans `src`.
4. Affichez la hiérarchie complète du répertoire `project` et des ses sous-répertoires, puis écrivez ce résultat dans `contents.txt`.
5. Créez une copie du répertoire `project` que vous nommerez `projectV2`. Supprimez le répertoire `project`.
6. Créez l'archive `pv2.tar` contenant l'ensemble du répertoire `projectV2`.

Exercice 3 – Premier script Shell

Afin d'automatiser l'exécution de commandes (comme par exemple la compilation d'un projet ou l'exécution d'un jeu de tests), il est possible de les rassembler dans un fichier. Ce type de fichier est appelé script. Placez l'ensemble des commandes écrites dans l'exercice 2 dans un script Shell, et exécutez-le. Le résultat est-il le même que dans l'exercice 2 ?

Exercice 4 – Shell en C

A l'aide de la fonction `system`, faites un programme C affichant le contenu de votre répertoire personnel.

Exercice 5 – Débogage

Compilez le programme `code-mystere.c` en utilisant l'option `-g` de `gcc`, puis déboguez-le à l'aide de `gdb` jusqu'à atteindre l'exécution normale du programme.

Rappel des commandes gdb :
`break fichier:ligne` – *ajout d'un point d'arrêt dans le code*
`run arg1 arg2 ...` – *exécution du programme*
`CTRL + c` – *envoi d'un signal d'interruption au programme*
`next` – *exécution de l'instruction suivante*
`continue` – *reprise de l'exécution du programme*
`print var` – *affichage du contenu d'une variable*
`backtrace` – *affichage de la pile d'appels des fonctions*
`up/down i` – *remontée/descente de i dans la pile d'appels*
`quit` – *arrêt du débogueur*

Exercice 6 – Optionnel : Shell en C (bis)

Écrivez le programme C répondant aux questions de l'exercice 2.

IN405 – Système d'exploitation
TD 2 – Système de fichiers (1/2)

S. Gougeaud

2017/2018

L'archive `code-td2.tar` *contient trois fichiers :*

- `se_fichier.h` – *fichier d'en-têtes contenant les prototypes de la 'bibliothèque' à écrire ;*
- `main.c` – *fichier principal de l'exercice 2, contenant différents scénarios utilisant la 'bibliothèque' ;*
- `main-opt.c` – *fichier principal de l'exercice optionnel 3, contenant différents scénarios utilisant les fonctions avancées de la 'bibliothèque'.*

Exercice 1 – Bibliothèque de fonctions d'E/S

L'objectif de ce premier exercice est de créer une bibliothèque d'entrée/sorties utilisant les appels systèmes liés au système de fichier. A cet effet, il vous est demandé d'écrire le contenu du fichier `se_fichier.c`, composé du corps des fonctions énoncées dans `se_fichier.h`. **Attention : vous devez respecter les prototypes du fichier d'en-tête.**

Pour compiler le fichier `se_fichier.c` et en obtenir une bibliothèque, vous devez utiliser les commandes suivantes :

```
$ gcc -c -fPIC se_fichier.c
$ ar rcs libsef.a se_fichier.o
```

Les fonctions de la bibliothèque sont alors contenues dans l'archive `libsef.a`. Pour utiliser les fonctions d'une bibliothèque dans un autre programme, le compilateur doit avoir accès à deux ressources : les fonctions compilées (ici `libsef.a`) et les prototypes de fonctions apportées par le fichier d'en-tête (ici `se_fichier.h`). En supposant que ces ressources sont situées dans le même répertoire que votre programme issu de `test.c`, sa compilation se fait à l'aide de la commande suivante :

```
$ gcc test.c -L. -lsef
```

L'option `-L` permet d'indiquer au compilateur un chemin supplémentaire pour la recherche de bibliothèque ; le chemin indiqué est le répertoire courant. L'option `-l` lie la bibliothèque sef (`libsef.a`) au programme.

1. Ecrivez la fonction `SE_ouverture()` qui ouvre le fichier dont le chemin est donné en paramètre. Attention à bien créer le fichier s'il n'existe pas.
2. Ecrivez la fonction `SE_fermeture()` qui ferme le fichier donné en paramètre.
3. Ecrivez la fonction `SE_suppression()` qui supprime le fichier donné en paramètre.
4. Ecrivez la fonction `SE_lectureCaractere()` qui lit le prochain caractère du fichier donné en paramètre. Vérifiez si le fichier a été ouvert avec les bonnes permissions.
5. Ecrivez la fonction `SE_ecritureCaractere()` qui écrit le caractère dans le fichier donné en paramètre. Vérifiez si le fichier a été ouvert avec les bonnes permissions.

Exercice 2 – Utilisation de la bibliothèque

A l'aide des fonctions de votre bibliothèque, remplissez les corps des fonctions suivantes du fichier `main.c`. Le programme issu de la compilation consiste en l'exécution de tests vérifiant le bon fonctionnement de ces fonctions.

1. Ecrivez la fonction `affichage()` ayant le même comportement que la commande `cat`.
2. Ecrivez les fonctions `copie()` et `deplacement()` ayant respectivement le même comportement que les commandes `cp` et `mv`.
3. Ecrivez la fonction `sontIdentiques()` qui retourne `1` si les fichiers sont identiques, et `0` sinon.

Exercice 3 – Optionnel : Amélioration de la bibliothèque

Vous allez maintenant ajouter des fonctions de lecture/écriture à votre bibliothèque. Pour ceci, décommentez les prototypes restants de `se_fichier.h`, et écrivez les corps correspondants dans `se_fichier.c`. Ces fonctions peuvent être testées grâce au programme issu de `main-opt.c`.

1. Ecrivez les fonctions de lecture/écriture d'une chaîne de caractère.
2. Ecrivez les fonctions de lecture/écriture d'un entier (comportement de scanf avec %d).
3. Ecrivez la fonction de lecture d'un mot (comportement de scanf avec %s)

IN405 – Système d'exploitation
TD 3 – Système de fichiers (2/2)

S. Gougeaud

2017/2018

Exercice 1 – Méta-données des fichiers

En utilisant les fonctions `stat` et/ou `lstat`, écrivez les fonctions répondant aux comportements suivants :

1. Affichage du type de fichier : socket (sock), lien symbolique (link), fichier régulier (file), device (devc), répertoire (repy), FIFO (fifo).
2. Affichage des permissions d'accès au fichier : rwxrwxrwx.
3. Affichage du propriétaire du fichier.
4. Affichage de sa taille

Exercice 2 – Lecture d'un répertoire

En utilisant les fonctions `opendir` et `readdir`, affichez le contenu d'un répertoire donné.

Exercice 3 – Implémentation de la commande `ls`

A partir des codes des deux exercices précédents, écrivez une fonction affichant le contenu d'un répertoire, et indiquant pour chaque item, son type, ses permissions en écriture, son propriétaire et sa taille dans le format suivant :

```
type rwxrwxrwx owner size name1
type rwxrwxrwx owner size name2
```

Exercice 4 – Optionnel : Edition de fichier

En utilisant les appels système `mmap`, `msync` et `munmap`, écrivez un programme prenant en argument un chemin de fichier, afin de l'éditer. L'édition consistera en un remplacement de chaque voyelle du fichier par un caractère '*'.

IN405 – Système d'exploitation
TD 4 – Processus

S. Gougeaud

Durée estimée : 6h

Exercice 1 – Création simple de processus

En utilisant la fonction `fork`, écrivez les programmes correspondant aux comportements suivants pour un processus père et son processus fils :

1. Affichage de 'Hello World!' pour les deux processus.
2. Affichage de 'Mon PID est ... et celui de mon père/fils est ... !'.
3. Le processus fils choisit aléatoirement un nombre entre 1 et 50, l'affiche, puis le communique à son père qui l'affiche à son tour.

Exercice 2 – `sleep()` & `wait()`

Écrivez le programme correspondant à l'énoncé suivant : le processus père crée 10 processus fils et attend qu'ils se terminent. Chaque fils attend un nombre de secondes choisi aléatoirement entre 1 et 10, affiche son PID puis se termine. Le processus père affiche à chaque terminaison, le PID du processus fils qui a terminé son exécution.

Exercice 3 – Création multiple de processus

Écrivez les programmes correspondants aux énoncés suivants, avec m et n, deux entiers donnés au lancement du programme :

1. Le processus père crée m fois n processus fils.
2. Le processus père crée m processus fils, puis chaque processus fils crée n processus petit-fils.
3. Le processus père crée n processus fils, puis chaque processus fils crée n processus petit-fils, puis chaque processus petit-fils etc., ceci m fois.

Pour chacun des énoncés, calculez, à l'aide du programme, le nombre total de processus créés.

Exercice 4 – Temps d'exécution

A l'aide de la fonction `times()`, écrivez le programme correspondant à l'énoncé suivant : le processus père crée un processus fils qui liste le contenu d'un répertoire donné en argument (à l'aide de la commande ls). Une fois l'exécution du fils terminée, le père affiche le temps d'exécution du processus fils (et donc de la commande ls). **Attention :** pour éviter un temps d'exécution quasi nul, n'hésitez pas à lister **récursivement** le répertoire.

Exercice 5 – Envoi de signal

A l'aide de la fonction `kill()` et des signaux `SIGSTOP` et `SIGCONT`, écrivez le programme correspondant à l'énoncé suivant : le processus père crée un processus fils qui compte de 1 à 5 (un affichage par seconde). Trois secondes après avoir créé son processus fils, le père met en pause le fils, attend cinq secondes puis le relance. Que se passe-t-il si le signal `SIGINT` est envoyé au lieu de `SIGSTOP` ?

Exercice 6 – Premiers pas avec les tubes

L'objectif de l'exercice est d'apprendre à utiliser les tubes anonymes et nommés à travers le transfert de simples données entre deux processus. Pour les énoncés suivants, implémentez une version en utilisant les tubes anonymes, et une seconde avec les tubes nommés :

1. Transfert d'une chaîne de caractères de taille 16 max (exemple : "Hello World!").
2. Transfert d'une paire d'entiers.

Exercice 7 – Optionnel : Gestion de signal

A l'aide de la fonction `sigaction()`, écrivez le programme correspondant à l'énoncé suivant : le processus père crée un processus fils qui compte de 1 à 12 (un affichage par seconde). Un signal `SIGUSR1` est envoyé par le père au fils à 3, 5 et 8 secondes. A la réception de ce signal, le processus fils affiche la phrase 'debug: x' avec x la valeur du compteur.

Indications :

- `sigaction()` demande en argument une structure `sigaction` composé des champs `sa_handler` et `sa_flags`.
- Le premier champ est un pointeur de fonction décrivant le comportement à adopter lors de la réception du signal (aussi appelé gestionnaire de signal).
- Le second champ **doit** être initialisé à `SA_ONSTACK` pour éviter la terminaison du processus (comportement par défaut de `SIGUSR1`).

- Pour obtenir la valeur du compteur à partir du gestionnaire de signal, il faut que ce compteur soit déclaré en variable globale.

IN405 – Système d'exploitation
TD 5 – Thread

S. Gougeaud

2017/2018

L'archive `code-td5.tar` *contient un fichier :*

- `reduc.c` – *fichier source de l'exercice 2, contenant les prototypes de fonction et les structures à utiliser.*

Exercice 1 – Création de thread

En utilisant la fonction `pthread_create()`, écrivez un programme où le processus principal créé un thread pour chacun des comportements suivants :

1. Affichage de 'Hello World!'.
2. Affichage d'un entier aléatoire généré par le processus principal.
3. Affichage d'un entier aléatoire généré par le thread qui sera aussi affiché par le processus principal.
4. Affichage de la moyenne d'un tableau de 5 entiers générés aléatoirement par le processus principal.
5. Affichage de la moyenne d'un tableau de n entiers générés aléatoirement par le processus principal.

Exercice 2 – Mécanisme de réduction pour le calcul

L'objectif de l'exercice est d'offrir à l'utilisateur une bibliothèque de fonctions et structures permettant le traitement d'un 'grand' tableau dans un environnement multi-threadé. L'exercice se découpe en plusieurs parties :

1. Programme principal – l'exécution du programme se fait par la ligne de commande suivante :

```
$ ./reduction m n opcode
```

L'argument `m` défini le nombre de threads générés par le programme principal, `n` la taille du tableau et `opcode` l'opération à réaliser sur le tableau. L'opcode Le programme doit dans un premier temps créer le tableau et générer les entiers le composant (entre 1 et 100), puis créer les threads. Il affiche, une fois l'exécution des threads terminée, le résultat obtenu.

2. Structure message – l'argument à la fonction de thread est une structure comportant quatre champs : le tableau d'entier (dans sa totalité), les indices de début et fin de traitement (partie du tableau que le thread doit traiter) et le résultat (renseignée par le thread à la fin de son exécution).

3. Détermination de l'opération – l'`opcode` est analysé par le programme principal, et ce dernier sélectionne alors l'opération à réaliser. Le tableau suivant représente les fonctions disponibles et leurs opcodes respectifs.

Code	Nom
+	somme
/	moyenne
M	max
m	min

4. Exécution du thread – chaque thread effectue sur la partie du tableau qu'il doit traiter, la fonction indiquée par l'opcode, puis retourner le résultat local.

5. Pour aller plus loin – vous pouvez gérer les arguments du programme avec la fonction `getopt()` (section 3 du manuel). La fonction `getopt()` permet l'analyse et le décodage des arguments d'un programme en utilisant des options. Il est alors possible de gérer des arguments obligatoires et/ou optionnels, quelque soit l'ordre dans lequel ils sont renseignés. L'exécution du programme se fera par la ligne de commande suivante :

```
$ ./reduction -t m -s n -o opcode
```

On supposera que l'argument '`-t`' pour le nombre de threads est optionnel, et les deux autres obligatoires. Dans le cas où le nombre de threads n'est pas renseigné, ce nombre vaudra 4.

IN405 – Système d'exploitation
TD 6 – Cohérence des données et synchronisation

S. Gougeaud

2017/2018

Exercice 1 – Section critique dans la réduction

Ré-implémentez l'exercice '**Mécanisme de réduction pour le calcul**' en utilisant une **SEULE** variable accessible par tous les threads pour le calcul du résultat. L'adresse de cette variable sera contenue dans la structure message.

Exercice 2 – Mécanisme de barrière

L'objectif de l'exercice est d'offrir à l'utilisateur un mécanisme de barrière entre n threads réutilisable. Il est proposé de l'implémenter en suivant plusieurs étapes :

1. Barrière avec 3 threads – la synchronisation est effectuée sur 3 threads, les deux premiers attendent que le troisième les libère.
2. Barrière avec n threads – les $(n-1)$ premiers threads attendent que le dernier les libère.
3. Ré-utilisabilité – la barrière peut être utilisée plusieurs fois dans le programme (avec le même nombre de threads se synchronisant).

Exercice 3 – Petits mots doux discrets

L'objectif de l'exercice est de faire passer un message (entier) d'un thread à l'autre en passant par n interlocuteurs. Chaque interlocuteur i peut transmettre/recevoir un message à/de ses voisins directs $(i-1)$ et $(i+1)$. Soit un nombre choisi aléatoirement entre 1 et 100 par le thread 1, faites-en sorte que le dernier thread le reçoive et l'affiche. Vérifiez que l'affichage est correct.

IN405 – Système d'exploitation
TD 7 – Tubes

S. Gougeaud

2017/2018

Exercice 1 – Premiers pas avec les tubes

L'objectif de l'exercice est d'apprendre à utiliser les tubes anonymes et nommés à travers le transfert de simples données entre deux processus. Pour les énoncés suivants, implémentez une version en utilisant les tubes anonymes, et une seconde avec les tubes nommés :

1. Transfert d'une chaîne de caractères de taille 16 max (exemple : "Hello World!").
2. Transfert d'une paire d'entiers.

Exercice 2 – Mécanisme de barrière

L'objectif de l'exercice est d'offrir à l'utilisateur un mécanisme de barrière entre n processus issus d'un même processus père, à l'aide de tubes (les mutex et conditions n'étant pas disponibles en mémoire non partagée). Chaque processus fils attend un nombre aléatoire de secondes compris entre 1 et 10, puis entre dans la barrière. Le père agit comme coordinateur, attendant que tous les fils soient dans la barrière pour les libérer.

Exercice 3 – Affichage informatisé

L'objectif de l'exercice est de faire passer un message à un processus indépendant (ne faisant pas partie de la même hiérarchie de processus), pour qu'il puisse l'afficher sur le terminal. L'exercice se découpe en plusieurs parties :

1. Programme 'affichage' – Ce programme doit afficher sur le terminal, toutes les 2 secondes, l'un des messages qu'il possède dans sa base de données (tableau de chaînes de caractère). Les messages sont lus les uns après les autres, et la lecture reprend au début de la base si on vient de lire la dernière entrée. Initialement, la base est constituée de deux messages :
   - "Bonjour, ceci est un affichage informatisé" ;

- "dd/MM/yy – hh:mm:ss" pour le jour et l'heure (`man gettimeofday` et `man localtime`).

2. Réception d'un signal de 'affichage' – A la réception du signal, le programme 'affichage' peut, en fonction de sa nature, soit ajouter à sa base la nouvelle information présente dans le tube, soit remettre la base dans sa configuration initiale ie. constitué des deux messages décrits précédemment. Pour ceci, vous pouvez utiliser les signaux `SIGUSR1` et `SIGUSR2`.

3. Programme 'envoi' – Ce programme doit envoyer au programme affichage des signaux en fonction de ses arguments. Il possède trois arguments : l'action à réaliser (remise à zéro ou ajout de message), le PID du programme affichage (pour ceci, l'afficher en début d'exécution) et le message à ajouter si besoin.

4. Quelques conseils :

   - La réception d'un signal entraînant une interruption de l'exécution normale du processus, la fonction 'handler' ne peut accéder qu'aux variables globales du programme : stockez le tableau (et autre(s) variable(s) NECESSAIRE(S)) dans des variables globales.
   - Les fonctions `open()` sur des tubes nommés sont bloquantes tant qu'il n'est pas ouvert dans les deux directions (lecture et écriture), n'ouvrez le tube dans le programme 'affichage' que si c'est nécessaire.
   - Le programme 'affichage' contient une boucle affichant périodiquement les messages de son tableau. Vous pouvez choisir de faire une boucle infinie ou une boucle se répétant un nombre défini de fois (20, 50, 100, etc.). Dans tous les cas, le raccourci `CTRL+c` envoie un signal de terminaison au processus s'exécutant au premier plan dans le terminal.

5. Pour aller plus loin, faites que le programme 'envoi' puisse terminer l'exécution du programme 'affichage' à l'aide d'un signal.

IN405 – Système d'exploitation
TD 8 – Exercices avancés

S. Gougeaud

2017/2018

Exercice 1 – Passage de jeton dans un réseau distribué

L'objectif de l'exercice est de créer un réseau de processus indépendants où la communication est assurée par des tubes. Un jeton, possédant une durée de vie définie, parcourt ce réseau jusqu'à que ce que sa vie se termine. Vous pouvez supposer le réseau présenté figure 1 pour vos tests.

L'exercice s'effectue en plusieurs étapes :

1) Construction du graphe

Chaque noeud du graphe correspond à un flux d'exécution d'un programme, et chaque arête à l'existence d'un moyen de communication bi-directionnel entre ces deux flux. Implémentez le programme correspondant à ce graphe à l'aide de processus pour les flux d'exécution et de tubes anonymes pour les moyens de communication. *On dit que le noeud 0 est le processus père du programme.*

2) Travail d'un noeud

Pour exécuter une instruction, un noeud du graphe doit détenir un jeton. L'instruction en question est l'affichage de la phrase "Je suis i et je possède le jeton !" dans laquelle i est le numéro du noeud. Une fois l'instruction effectuée, le jeton est envoyé aléatoirement à un autre noeud. Implémentez la fonction qui se met en attente d'un jeton, fait l'affichage puis renvoie le jeton.

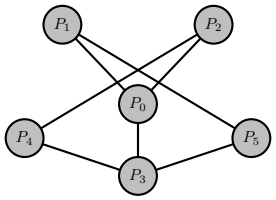

Figure 1: Réseau distribué test

3) Initialisation et mise à jour du jeton

On suppose que le jeton est détenu au début du programme par le noeud 0, et qu'il possède une durée de vie égale à 20. A chaque fois que le jeton est utilisé (ie. qu'une instruction est effectuée par le noeud possédant le jeton), sa durée de vie décrémente d'un point. Implémentez l'initialisation du jeton par le processus père, et la mise à jour du jeton après chaque affichage réalisé.

4) Fin de vie du jeton et destruction du graphe

Lorsque la durée de vie du jeton est nulle, le programme s'arrête. Il faut alors prévenir l'ensemble des noeuds du graphe qu'il faut terminer l'exécution. A cet effet, le noeud recevant le jeton dont la durée de vie est nulle, prévient tous les noeuds qu'il a dans son entourage que le programme se termine (par un message contenant l'entier -1 par exemple). A la réception de ce message, chaque noeud préviendra à son tour tous les noeuds de son entourage, ainsi de suite jusqu'à ce que tous les noeuds aient été prévenus et soient terminés. Implémentez ce phénomène.

Exercice 2 – Chasse au trésor dans un fichier partagé

L'objectif de l'exercice est de créer un jeu dans lequel un chasseur de trésor creuse à différents endroits d'une île afin d'emplir ses poches de richesse. Le programme est composé de trois acteurs différents :

- L'île, qui connaît la topographie des lieux et sait où se trouve le trésor.
- Le chasseur, qui cherche à récupérer le trésor en creusant aléatoirement (ou non ?) sur l'île.
- La volcan, qui peut à tout moment détruire l'île, son trésor et ses occupants s'il y en a.

L'appel au programme se fait par l'appel suivant :

```
$ ./hunt [i/h/v] [arg1] [arg2]
```

L'argument `[i/h/v]` détermine le rôle d'acteur du processus créé, entre l'île (island), le chasseur (hunter) et le volcan (volcano). Les arguments `arg1` et `arg2` dépendent de l'acteur créé. La suite de l'énoncé décrit le comportement de chaque acteur.

1) L'île – *The island*

L'île est constituée de $m \times n$ cases où chaque case est l'un des éléments indiquée dans la table 1. La constitution initiale de la carte se fait à la création du processus, où une case possède une probabilité p d'être de l'élément indiqué. Creuser une case pour chercher le trésor demande un apport e en énergie devant être fournie par le chasseur. La topographie de la carte est seulement connue de l'île.

type	p	e
eau	10	1
sable	50	1
roche	10	3
palmier	15	2
épave	10	5
volcan	5	4

Table 1: Paramètres de l'île

L'île partage avec le chasseur un plan sommaire où sont référencées les cases ayant déjà été creusées. Ainsi, une case ne peut pas être creusée plus d'une fois.

L'emplacement du trésor est défini à la création du processus, mais ne peut pas se trouver sur une case d'eau ou de volcan. Si le trésor a été trouvé par le chasseur ou si le volcan entre en éruption, alors l'île s'éteint.

L'appel au programme pour la création de l'île est : `$ ./hunt i m n`

2) Le chasseur – *The hunter*

Le chasseur possède une énergie initiale E. Il demande à son initialisation la taille de l'île et commence à creuser là où des recherches n'ont pas encore été effectuées. S'il trouve le trésor, alors il repart victorieux. Si un autre chasseur trouve le trésor avant lui, alors il repart défait. Si son énergie tombe à 0, alors il meurt d'épuisement. Si le volcan entre en éruption, alors il meurt tout court.

L'appel au programme pour la création du chasseur est : `$ ./hunt h E`

3) Le volcan – *The volcano*

Le volcan a pour seule vocation d'entrer en éruption. Lorsqu'il est créé, l'île et les chasseurs sont détruits, brûlés au 42ème dégré.

L'appel au programme pour la création du volcan est : `$ ./hunt v`

4) Consignes d'implémentation

- La topographie de la carte est uniquement possédée par l'île.
- La carte sommaire, indiquant quelles cases ont déjà été creusées est partagée en mémoire dans un fichier. Pour ceci, aidez-vous de l'exercice sur `mmap()` du TD 3.
- La communication demandée est bi-directionnelle : dans un premier temps, le chasseur indique quelle case il cible, puis l'île lui répond en lui donnant le montant d'énergie qu'il a perdu et/ou s'il a trouvé le trésor. Il faut donc au moins un tube pour les messages à destination de l'île, et un autre tube pour les messages à destination d'**un** chasseur.

- Pour permettre à l'île de 'tuer' les chasseurs se trouvant en son sein lorsque le volcan entre en éruption, il est préférable de garder en mémoire une liste des chasseurs.