



Systemes d'exploitation


— Travaux pratiques —

Alain Lebreton

SPÉCIALITÉ INFORMATIQUE, ENSICAEN, FRANCE
Version 2.3, 10 octobre 2018

EXEMPLIER DU COURS : HTTPS://GITHUB.COM/ALAINLEBRET/OS
--

Par leurs précieuses remarques, les personnes suivantes ont aussi participé à l'amélioration de ce document : Thomas Gougeon, Patrick Lacharme et Mathieu Valois.

Les illustrations d'en-tête sont libres de droits  excepté les suivantes :

- table des matières : K. Martin (CC BY-NC-SA 2.0)
- TP sur les *threads* : Cburnett (CC BY-SA 3.0)

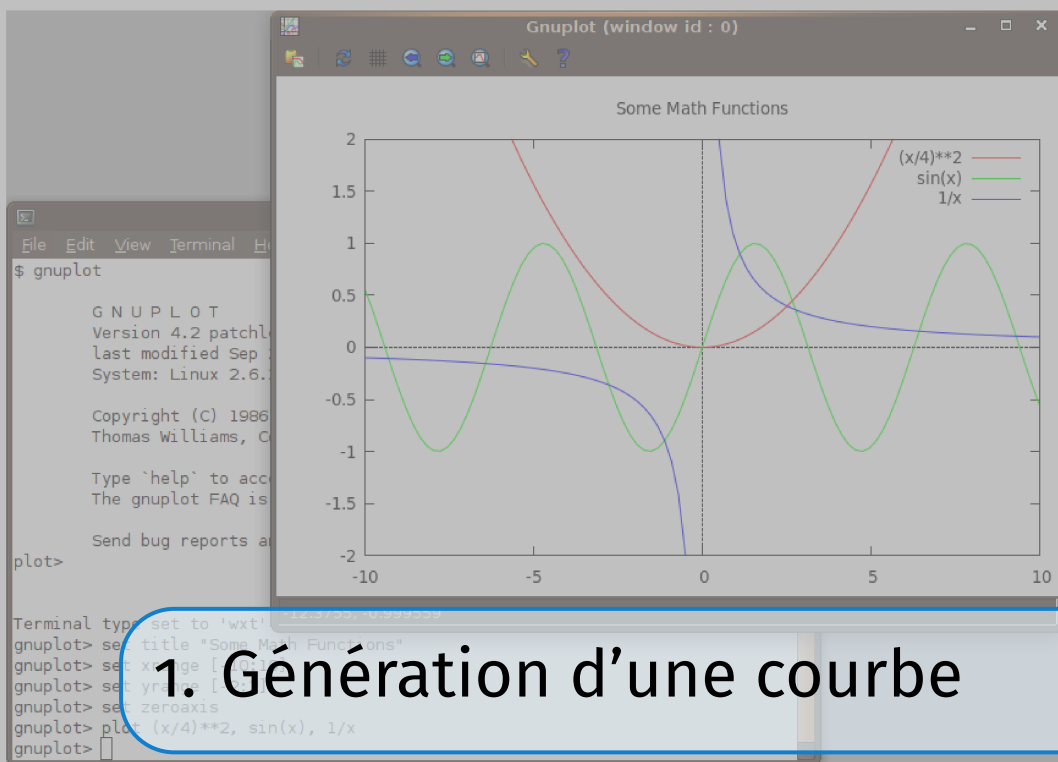


Contenu

1	Génération d'une courbe	7
1.1	Présentation	7
1.2	Travail à réaliser	8
2	Génération périodique de données	9
2.1	Présentation	9
2.2	Travail à réaliser	9
3	Criblage par processus filtrants	13
3.1	Présentation	13
3.2	Exercice préparatoire	14
3.3	Travail à réaliser	14
4	Produit matriciel multithreadé	15
4.1	Présentation	15
4.2	Travail à réaliser	15
5	A License to Kill	17
5.1	Présentation	17
5.2	Travail à réaliser	18

Vos codes sources doivent respecter les règles de codage spécifiées dans le document “Linux kernel coding style” disponible à l’adresse suivante :
<https://www.kernel.org/doc/Documentation/process/coding-style.rst>

PENSEZ MODULARITÉ : PETITES FONCTIONS, MAIN() TRES COURT !



1. Génération d'une courbe

*There are only two hard things in
Computer Science: cache
invalidation and naming things.*

PHILIP KARLTON

Objectif

Mettre en œuvre les processus à l'aide des appels système `fork` et `exec`
(durée estimée : 1 séance)

1.1 Présentation

Le logiciel "gnuplot" (<http://www.gnuplot.info>) permet de représenter graphiquement des données numériques simples sous forme d'un diagramme, d'une courbe ou encore d'un histogramme. Il est accessible en ligne de commande à partir du terminal :

```
gnuplot -persist commandes.gp
```

où `commandes.gp` est un fichier de commandes gnuplot et l'option `-persist` permet à la fenêtre de visualisation de rester ouverte.


Voici un exemple de fichier de commandes permettant de visualiser la courbe du sinus cardinal sur l'intervalle $[0, 50]$:

```
set samples 500
set title "Sinus cardinal"
set xlabel "x"
set ylabel "sin(x)/x"
set xrange [0:50]
set border
set grid
plot sin(x)/x
```

1.2 Travail à réaliser

On se propose de créer un programme de tracé de courbe et qui attend en permanence que l'utilisateur indique une borne B permettant de définir l'intervalle de visualisation de la courbe (vous fixerez au choix un intervalle entre $[0, B]$ ou entre $[-B, B]$). Dès que la borne est fournie, le processus père crée un fichier `commandes.gp`, puis un processus fils chargé d'exécuter "gnuplot" de manière à afficher la courbe dans le nouvel intervalle, puis il se remet en attente d'une nouvelle borne.

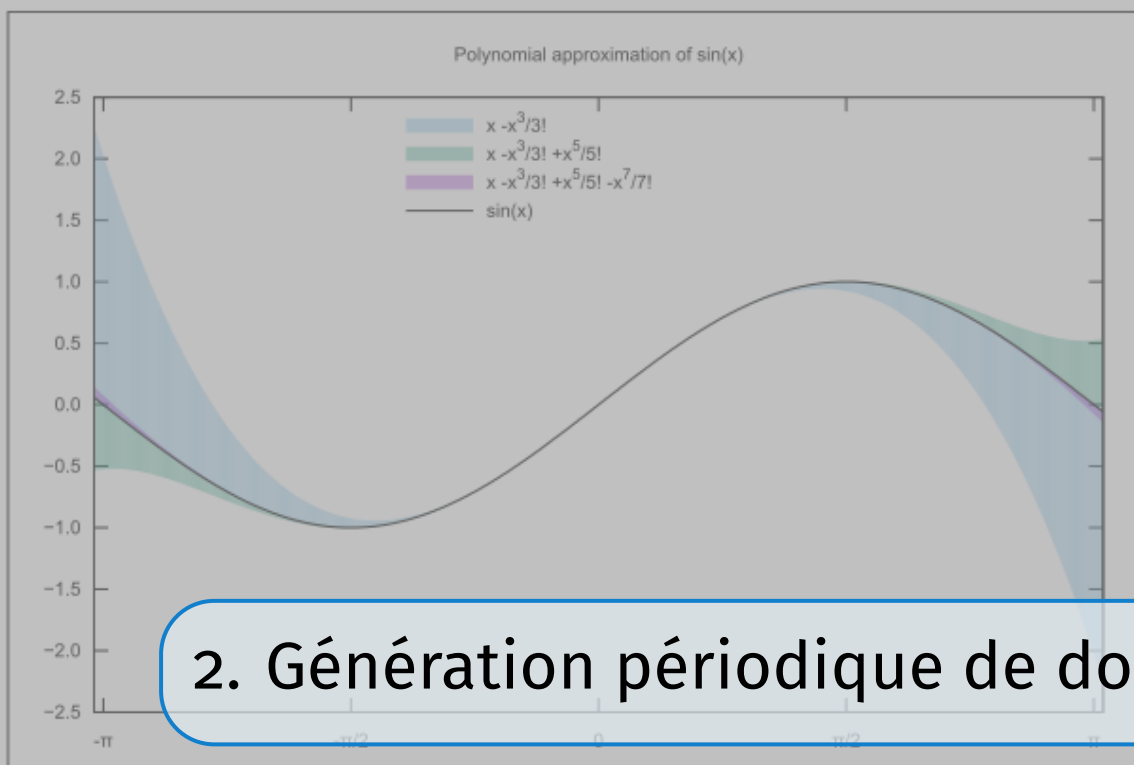
1. Créer un dossier TP1 dans le répertoire de travail OS puis s'y déplacer.
2. Coder le programme `ipplot` qui réalise les opérations précédemment décrites. Celui-ci sera construit à partir des fichiers `ipplot.c`, `ipplot.h` et `main.c`. La primitive `execlp()` pourra être utilisée afin de lancer "gnuplot".

 Afin de visualiser le mécanisme de recouvrement, ouvrez deux terminaux et dans l'un deux, entrez la commande `top -u utilisateur` qui affiche la charge processeur en temps réel pour l'utilisateur spécifié. Puis ajoutez un appel à la fonction `sleep()` dans le processus fils avant son appel à `execlp()`. Exécutez le programme et notez la création du processus fils, puis au bout du temps imparti, sa disparition et l'apparition du processus "gnuplot".

Livrable

En fin de séance, déposer sur la plateforme Moodle une archive du répertoire TP1 contenant :

1. Un fichier `README.txt` donnant la commande pour installer et exécuter le logiciel, et indiquant éventuellement vos commentaires.
2. Le fichier `commandes.gp` utilisé par gnuplot.
3. Le(s) fichier(s) source(s) `.c` et `.h`.
4. Le fichier `Makefile` permettant de construire tout le projet.



2. Génération périodique de données

*Once you understand how to write
a program get someone else to
write it.*

ALAN PERLIS

Objectif

Mettre en œuvre les signaux à l'aide l'appel système POSIX `sigaction`
(durée estimée : 1 séance)

2.1 Présentation

Dans ce TP, nous allons mettre en œuvre la gestion des signaux `SIGALRM` et `SIGCHLD`. `SIGALRM` est le signal émis par le noyau à un intervalle régulier que l'on peut définir. `SIGCHLD` quant à lui, est envoyé par un processus fils à son père lorsque le processus fils se termine ou est interrompu.

2.2 Travail à réaliser

1. Construire un dossier TP2 dans le dossier de travail OS puis s'y déplacer.
2. Proposer le programme `sin_cos` qui crée deux processus fils chargés d'afficher, dans la mesure du possible à tour de rôle, la valeur des *sinus* et *cosinus* d'angles incrémentés de dix degrés toutes les secondes. Les angles seront incrémentés par le traitant mis en place par chacun des fils.

Exemple d'affichage :

```
./sin_cos
Fils 1 (6837) : sinus(0)=      0.00
Fils 2 (6838) : cosinus(0)=    1.00
Fils 1 (6837) : sinus(10)=     0.17
Fils 2 (6838) : cosinus(10)=   0.98
```

```

Fils 1 (6837) : sinus(20)=      0.34
Fils 2 (6838) : cosinus(20)=   0.94
Fils 1 (6837) : sinus(30)=     0.50
Fils 2 (6838) : cosinus(30)=   0.87
Fils 1 (6837) : sinus(40)=     0.64
Fils 2 (6838) : cosinus(40)=   0.77
Fils 1 (6837) : sinus(50)=     0.77
Fils 2 (6838) : cosinus(50)=   0.64

```

Le programme sera construit à partir des fichiers `sin_cos.c`, `sin_cos.h` et `main.c`.

- Proposer le programme `sin_cos_g` qui crée deux processus fils chargés d'enregistrer chaque seconde et à chaque ligne dans les fichiers `sinus.txt` et `cosinus.txt` qu'ils auront préalablement créés, l'angle et son sinus (respectivement son cosinus) avec des incréments de dix degrés, le tout sur une période. Les fichiers seront au format reconnu par *gnuplot*¹ pour le tracé de courbes et contiendront alors les données sous la forme suivante :

```

x1 y1
x2 y2
...
xn yn

```

où x_i est une abscisse (l'angle) et y_i l'ordonnée (le sinus ou le cosinus) du point à afficher.

Le processus père, quant à lui, attend les signaux de fin de ses deux premiers fils. Il crée alors deux nouveaux fils qui exécutent *gnuplot* de manière à réaliser l'affichage.

Le programme sera construit à partir des fichiers que vous nommerez `sin_cos_g.c`, `sin_cos_g.h` et `main.c`.

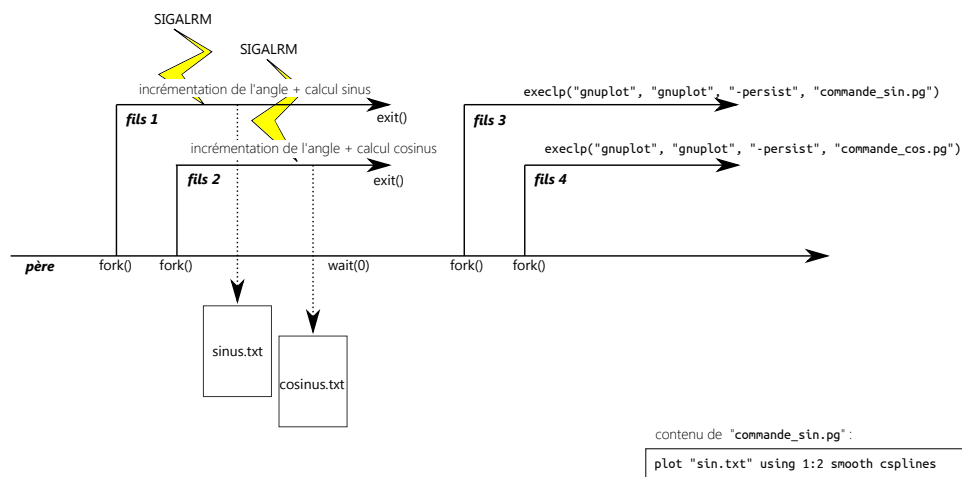


Figure 2.1: Représentation temporelle du programme `sin_cos_g`.

Livrable

En fin de séance, déposer sur la plateforme Moodle une archive du répertoire TP2 contenant pour chacun des deux programmes :

¹Un tutoriel sur *gnuplot* créé par Bernard Desgraupes peut être récupéré à l'adresse suivante : http://bdesgraupes.pagesperso-orange.fr/UPX/Tutoriels/presentation_gnuplot.pdf

-
1. Un fichier `README.txt` donnant la commande pour installer et exécuter le programme, ainsi que vos commentaires éventuels.
 2. Le(s) fichier(s) source(s) `.c` et `.h`.
 3. Le fichier `Makefile` permettant de construire tout le projet.



3. Criblage par processus filtrants

There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C.A.R. HOARE

Objectif

Appréhender la communication interprocessus par tubes anonymes
(durée estimée : 2 séances)

3.1 Présentation

Charles Antony Richard Hoare a proposé en 1978 de calculer les nombres premiers inférieurs à N^2 en utilisant $N+2$ processus qui reçoivent de leur prédécesseur une suite croissante de nombres dont le premier, n , est un nombre premier, et transmettent à leur successeur les nombres de cette suite, dans le même ordre, à l'exception de n et de ses multiples.

Nous nous proposons d'utiliser des tubes de communication afin de permettre aux processus de s'échanger les suites de nombres, puis de sauvegarder, en fin de compte, dans le fichier `prime_numbers.txt`, l'ensemble des nombres premiers inférieurs ou égaux à une valeur passée en argument au programme (`max_val`).

Pour ce faire, nous mettons en place le mécanisme suivant :

Un père va créer un fils f_1 puis va lui communiquer la suite des entiers compris entre 2 et `max_val` à l'aide d'un tube anonyme. De manière générale, un fils f_i reçoit comme premier entier un nombre premier n_i . Il le sauvegarde dans le fichier puis il filtre les entiers qui suivent :

- si l'entier est divisible par n_i il ne fait rien ;
- au premier entier $n_i + 1$ non divisible par n_i il crée lui-même un processus fils ainsi qu'un tube de communication avec celui-ci ; l'entier $n_i + 1$ ainsi que tous les autres entiers non divisibles par n_i sont alors envoyés à son fils.


Lorsque le père a terminé d'envoyer tous ses entiers, il envoie l'entier -1 à son fils puis attend la mort de ce dernier.

3.2 Exercice préparatoire

Réaliser un programme dans lequel un processus père envoie à son fils, par l'intermédiaire d'un tube anonyme, une suite de nombres entiers que le fils doit afficher sur la sortie standard. La réception du nombre -1 termine le fils.

3.3 Travail à réaliser

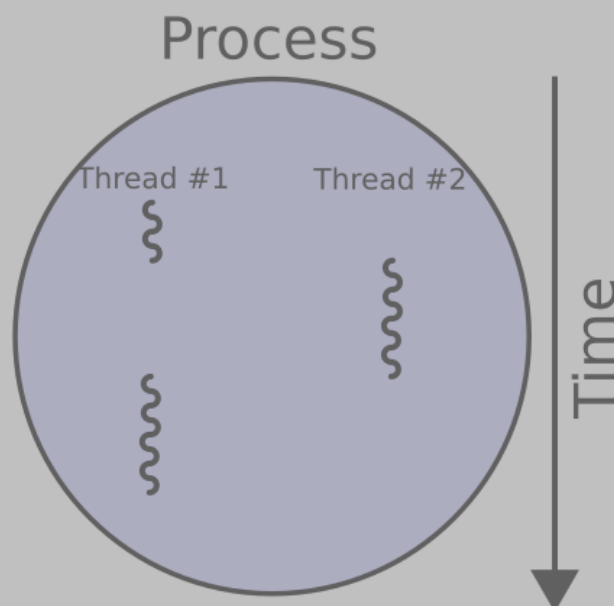
1. Construire un nouveau dossier TP3 dans le dossier OS puis s'y déplacer.
2. Coder le programme `fsieve` qui réalise les opérations précédemment décrites. Celui-ci sera construit à partir des fichiers `fsieve.c`, `fsieve.h` et `main.c`.

 Le fichier `prime_numbers.txt` étant partagé par le père et ses fils, il est nécessaire de faire appel à `fflush()` après chaque écriture dans le fichier.

Livrable

En fin de séance, déposer sur la plateforme Moodle une archive du répertoire TP3 contenant :

1. Un fichier `README.txt` donnant la commande pour installer et exécuter le logiciel, ainsi que vos commentaires éventuels.
2. Le(s) fichier(s) source(s) `.c` et/ou `.h`.
3. Le fichier `prime_numbers.txt` généré par votre logiciel.
4. Le fichier `Makefile` permettant de construire tout le projet.



4. Produit matriciel multithreadé

*Would you rather Test-First, or
Debug-Later ?*

ROBERT MARTIN

Objectif

Mettre en œuvre les *threads*
(durée estimée : 1 séance)

4.1 Présentation

Dans ce TP, nous allons mettre en œuvre les *threads* POSIX afin de réaliser des produits matriciels.

4.2 Travail à réaliser

1. Créer un dossier TP4 dans le dossier de travail OS puis s'y déplacer.
2. Proposer le programme `matrix_product` qui réalise le produit de deux matrices de doubles dont la représentation en mémoire est donnée par la figure ci-dessous, et qui est composé (au minimum) des fonctions suivantes :
 - a) `alloc_matrix()` qui permet d'allouer dynamiquement une matrice de `row` lignes et `column` colonnes.
 - b) `random_matrix()` qui affecte aléatoirement les coefficients d'une matrice de taille donnée.
 - c) `product_matrix()` qui effectue le produit de deux matrices de taille donnée.
 - d) `product_matrix_thread()` qui effectue le produit de deux matrices de taille donnée à l'aide de 4 *threads*. Chaque *thread* se chargera du calcul d'un quart de la matrice résultat.

Le programme sera construit à partir des fichiers `matrix_product.c`, `matrix_product.h` et `main.c`.

3. Comparer à l'aide du code ci-dessous, les temps d'exécution des deux fonctions précédentes (avec le même jeu de données). Dans quelles conditions peut-on espérer observer un gain de temps d'exécution ?

```

clock_t c_before, c_after;
time_t t_before, t_after;

c_before = clock();
t_before = time(NULL);

/* ----- */
/*  Calcul du produit matriciel  */
/* ----- */

t_after = time(NULL);
c_after = clock();

printf("\nClock_t -> %5.3f ticks (%f seconds)\n",
      (float) (c_after - c_before),
      (double) (c_after - c_before) / CLOCKS_PER_SEC);
printf("Time_t -> %5.3f seconds\n", difftime(t_after, t_before));

```

Livvable

En fin de séance, déposer sur la plateforme Moodle une archive du répertoire TP4 contenant :

1. Un fichier README.txt donnant la commande pour installer et exécuter les logiciels, ainsi que vos commentaires éventuels.
2. Le(s) fichier(s) source(s) .c et .h.
3. Le fichier Makefile permettant de construire tout le projet.

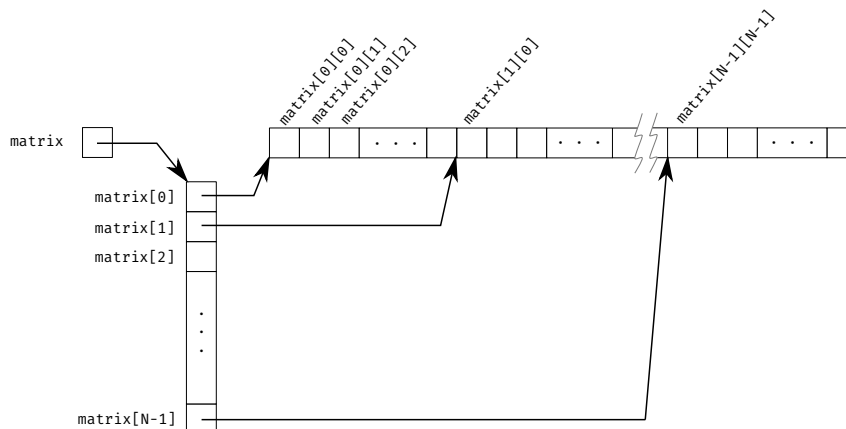


Figure 4.1: Représentation d'une matrice en mémoire



5. A License to Kill

If you're having trouble sounding condescending, get a Unix user to show you how.

SCOTT ADAMS

Objectif

Mettre en œuvre la communication interprocessus par mémoire partagée et gérer les problèmes de synchronisation

(durée estimée : 2 séances)

5.1 Présentation

Un officier traitant (*case officer*) reçoit les informations de ses agents à l'aide d'une boîte aux lettres que nous représenterons par une mémoire partagée. Les messages déposés par les agents sont chiffrés par une méthode simpliste.

Nous considérerons que notre histoire se déroule sur 24 heures (2 heures en réalité). Les agents déposent un message toutes les 5 à 15 min, soit toutes les 25 à 75 s dans la réalité. L'officier traitant, quant à lui, relève les messages toutes les 2 à 5 min (10 – 25 s). Les messages déposés ne sont pas tous des messages importants et on considérera qu'en moyenne seuls 10 % le sont, les autres étant considérés comme des messages "trompeurs".

Vous disposez des programmes suivants :

- `case_officer` qui réalise le fonctionnement de l'officier traitant et s'exécute sans arguments depuis un terminal. Il est chargé de la création de la boîte aux lettres, puis à chaque nouvelle vérification de la boîte, il affiche sur sa sortie standard : le caractère ✕ si le message est sans importance et le caractère ✓ dans le cas contraire¹. Le processus est stoppé s'il reçoit un signal SIGINT (<Ctrl-C>).

¹Ces caractères font partie des polices *Kelvinch* et *Everson Mono* présentes sur Ubuntu

- `agent` qui réalise le fonctionnement d'un agent source. Le programme est aussi lancé sans arguments depuis un terminal. Il se charge de déposer dans la boîte aux lettres des messages importants ou "trompeurs". À chaque nouveau dépôt, il affiche sur sa sortie standard le caractère `␣`. Le processus est automatiquement stoppé en fin de séance, ou encore s'il reçoit un signal `SIGINT` (`<Ctrl-C>`). Un agent peut subir des blessures lorsqu'il reçoit des signaux : blessures minimales pour les projectiles `SIGUSR1` (il affiche alors le caractère `☹`) ou plus sérieuses avec les projectiles `SIGUSR2`, auquel cas il affiche `☹☹`.
- `killer_agent` qui réalise le fonctionnement d'un agent source capable de riposter violemment. Le programme est aussi lancé sans arguments depuis un terminal. Lorsqu'il est blessé, ce type d'agent répond en envoyant un projectile `SIGTERM` au *singleton* l'ayant attaqué, puis il affiche entre parenthèses les caractères `☹☹` et `→`, suivis du PID de l'attaquant.

5.2 Travail à réaliser

Créer un dossier TP5 dans le dossier de travail OS puis s'y déplacer.

1) Interception

Votre mission consiste dans un premier temps à intercepter les messages qui sont déposés dans la boîte. Toutefois, vous ne partez pas sans indices. Une analyse de la boîte vous a renseigné sur le format des messages et la manière dont ils sont stockés dans la boîte. Ainsi un message est une structure formée de deux tableaux de caractères :

- un tableau de 20 caractères indiquant l'agent ayant émis le message ;
- un tableau de 256 caractères pour le contenu du message lui-même.

La boîte quant à elle est structurée de la manière suivante :

- un tableau de trois messages ;
- deux entiers dont vous ne connaissez pas le rôle.

Enfin, une indiscretion vous a révélé que la méthode de chiffrement utilisée par les agents de ce service était très certainement un chiffrement par décalage très simple.

Réalisez le programme `singleton.c` (un *singleton* est un agent espion non attaché à un service) permettant d'intercepter et de décoder les messages. Par la suite, seuls les messages importants seront décodés et conservés. Pour vos tests, vous pourrez lancer une instance du programme `case_officer` et trois instances du programme `agent` dans quatre terminaux séparés.

2) Licence to kill

Vous souhaitez supprimer un des agents afin de prendre sa place et envoyer de faux messages. Proposez le programme `singleton2.c` qui améliore votre ancien *singleton* en conséquence.

3) Nouveau réseau

Proposer les programmes `case_officer2.c` et `agent2.c` permettant de gérer un réseau d'espionnage de même type, mais plus performant que celui qui a été démasqué.

Livrable

En fin de séance, déposer sur la plateforme Moodle une archive du répertoire TP5 contenant :

1. Un fichier `README.txt` donnant la commande pour installer et exécuter le logiciel, ainsi que vos commentaires éventuels.
2. Le(s) fichier(s) source(s) `.c` et `.h`.
3. Le fichier `Makefile` permettant de construire tout le projet.

Bon courage !

Annexes

Mémoire partagée

La mémoire partagée, les files d'attente de messages et les sémaphores forment une suite de mécanismes de communication interprocessus (IPC) disponibles sur les systèmes Unix. Dans le cas de la mémoire partagée, le système fournit un segment de mémoire partagée qu'un processus appelant peut projeter ("mapper") sur son espace d'adressage personnel. Après cette opération, le segment sera considéré comme n'importe quelle autre partie de l'espace d'adressage du processus.

La mémoire partagée est le plus rapide des mécanismes permettant de transmettre des données conséquentes entre deux processus sur un même système hôte. Pour chacun des mécanismes IPC, il existe deux séries d'appels, les appels System V traditionnels et les appels POSIX plus récents. Ici, nous ne considérons que les appels POSIX.

Des exemples de programmes réalisant des processus "producteur" et "consommateur" qui communiquent à l'aide d'une mémoire partagée POSIX sont fournis à la fin de cette section.

Mécanisme POSIX

Les appels de mémoire partagée POSIX sont basés sur la philosophie Unix qui considère que les opérations d'entrée/sortie correspondent à des opérations sur un fichier. Ainsi, un objet de type mémoire partagée POSIX correspond à un fichier projeté en mémoire (voir figure 5.1). Les fichiers de mémoire partagée POSIX sont accessibles à partir d'un système de fichiers *tmpfs* monté sur `/dev/shm`.

Les fichiers de mémoire partagée individuels sont créés sous `/dev/shm` à l'aide de l'appel système `shm_open()`. Il existe deux primitives permettant ensuite d'ouvrir puis de libérer une mémoire partagée POSIX : `shm_open()` et `shm_unlink()` qui sont analogues à l'ouverture et à la fermeture pour les fichiers.

D'autres opérations sur la mémoire partagée POSIX sont nécessaires avant de l'utiliser et elles mettent en œuvre les primitives `ftruncate()`, `mmap()` et `munmap()`. La figure 5.2 montre les étapes pour la création de la mémoire partagée dans le cas

- Ⓡ Un programme utilisant des appels d'accès à une mémoire partagée POSIX doit en principe être lié avec l'option `-lrt` sous Linux (pas sous Mac OS X).

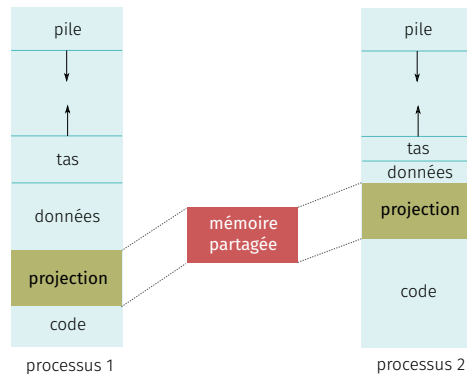


Figure 5.1: Mécanisme de “projection” en mémoire

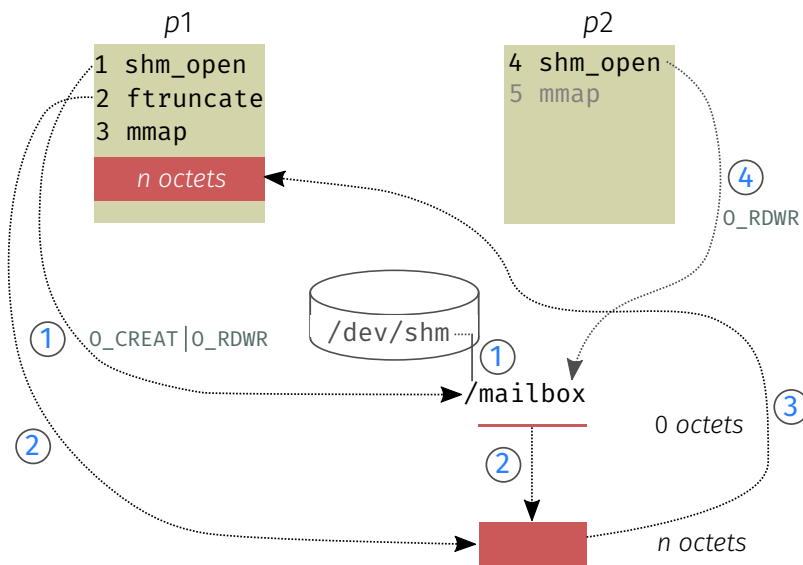


Figure 5.2: Étapes de création et accès de la mémoire partagée pour le réseau d’espionnage.

Primitive shm_open()

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_open(const char *name, int oflag, mode_t mode);
```

shm_open() correspond à l’appel système d’ouverture des fichiers (open()). La primitive ouvre un objet de mémoire partagée POSIX et le met à la disposition du processus appelant par l’intermédiaire du descripteur de fichier retourné. Le premier paramètre, name, est le nom de l’objet de mémoire partagée et est de la forme “/un_nom” (commence toujours par le caractère ‘/’ mais n’en contient pas d’autre). Les autres paramètres sont identiques à ceux que l’on retrouve dans l’appel à open(). Par exemple, le processus à l’origine de la création de la mémoire et ayant la possibilité de lire et d’écrire dans cette mémoire utilisera une valeur de oflag égale à O_CREAT | O_RDWR, alors qu’un processus qui se contente d’y accéder pour y lire et y écrire en considérant cette dernière déjà créée, utilisera une valeur de oflag égal à O_RDWR.

- Ⓡ Une fois créée, le fichier de mémoire partagée peut être listé (`ls /dev/shm`) et son contenu visualisé (`cat /dev/shm/mem_morpion`).

Primitive `shm_unlink()`

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
int shm_unlink(const char *name);
```

`shm_unlink()` supprime l'objet de mémoire partagée POSIX précédemment créé. Le nom est celui de l'objet mémoire partagée tel que décrit ci-dessus. Cette primitive doit absolument être appelée avant la terminaison du programme. En cas de processus mettant en œuvre une séquence infinie, il est conseillé de l'appeler depuis un traitant de signal que vous aurez préalablement défini, par exemple à l'aide du signal `SIGINT` (CTRL-C).

Autres primitives utilisées

Primitive `ftruncate()`

```
#include <unistd.h>
#include <sys/types.h>
int ftruncate(int fd, off_t length);
```

Lorsqu'une mémoire POSIX partagée est créée, elle est de taille nulle. L'utilisation de `ftruncate()` permet de l'agrandir de `length` octets. L'appel système `ftruncate()` retourne alors l'objet référencé par le descripteur de fichier `fd`. La primitive renvoie zéro en cas de succès et sinon la variable `errno` est définie sur la cause de l'erreur.

Primitive `mmap()`

```
#include <sys/mman.h>
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

L'appel système `mmap()` permet de projeter un objet de type mémoire partagée POSIX² dans l'espace d'adressage virtuel du processus appelant.

Le paramètre `addr` spécifie l'adresse à laquelle la projection sera réalisée et dans la plupart des cas on le positionnera à `NULL`. `length` définit la taille de la mémoire partagée qui doit être projetée. Pour simplifier les choses, nous projeterons l'objet entier et sa longueur sera celle de la mémoire partagée (ce qui n'est pas une obligation). `prot` peut prendre les valeurs : `PROT_EXEC`, `PROT_READ`, `PROT_WRITE` ou `PROT_NONE`. `PROT_EXEC` signifie que les pages projetées peuvent être exécutées et `PROT_NONE` qu'elles sont inaccessibles. Ces valeurs n'ont pas de sens pour un objet de type mémoire partagée. Nous allons donc nous contenter des valeurs `PROT_READ` ou `PROT_WRITE`. Le paramètre `flags` prendra dans notre cas la valeur `MAP_SHARED`, ce qui signifie que les mises à jour de la mémoire partagée projetée seront immédiatement visibles par tous les autres processus. `fd` est le descripteur de fichier pour la mémoire partagée retourné lors de l'appel à `shm_open()`. Enfin, `offset` est l'emplacement dans l'objet mémoire partagée à partir duquel la projection commence ; nous utiliserons la valeur 0 pour ce dernier et projeterons l'objet mémoire partagée en commençant dès le début.

En cas de succès, `mmap` retourne un pointeur vers l'emplacement où l'objet de mémoire partagée a été projeté. En cas d'erreur, elle retourne `MAP_FAILED` (`(void *)-1`) et `errno` est définie sur la cause de l'erreur.

²Mais pas seulement comme nous l'avons vu dans le cours (chapitre "Fichiers").

Primitive munmap()

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
```

La primitive `munmap()` dé-projette l'objet mémoire partagée placé à l'emplacement pointé par `addr` et ayant la taille `length`. En cas de succès, `munmap` retourne 0. En cas d'erreur, elle renvoie -1 et `errno` est définie sur la cause de l'erreur.

Exemple

```
/**
 * @file producteur_mem.c
 *
 * Processus producteur qui écrit indéfiniment un entier dans une mémoire
 * partagée, entier qu'il incrémente ainsi que sa racine carrée.
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>

#define EVER ;;
#define CHEMIN_MEMOIRE "/mem_racine" /* doit commencer impérativement par ↵
    "/" */

/** Structure utilisée dans le segment de mémoire partagée */
struct memoire_t {
    int valeur; /*!< valeur entière */
    double racine; /*!< racine carrée */
};

void traiter_erreur(char *message)
{
    fprintf(stderr, "%s", message);
    exit(EXIT_FAILURE);
}

/**
 * Traitant pour sortir de la boucle infinie et quitter en libérant la mé↵
    moire.
 */
void traiter_sigint(int signum)
{
    if (shm_unlink("CHEMIN_MEMOIRE") < 0) {
        traiter_erreur("Erreur lors de l'appel à shm_unlink\n");
    }
}
```

```

    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[]) {
    int entier;
    struct memoire_t *memoire; /* pointeur sur le segment mémoire "projeté" */
    int descripteur_memoire; /* descripteur du segment de mémoire partagée */
    int taille_memoire;

    taille_memoire = (1 * sizeof(struct memoire_t));

    signal(SIGINT, traiter_sigint); /* préférez la version POSIX vue en cours ↵
        */

    /* - 1 - crée le segment de mémoire partagée comme un fichier */
    descripteur_memoire = shm_open(CHEMIN_MEMOIRE, O_CREAT|O_RDWR, S_IRUSR|↵
        S_IWUSR);
    if (descripteur_memoire < 0) {
        traiter_erreur("Erreur lors de l'appel à shm_open\n");
    }
    fprintf(stderr, "Objet de mémoire partagée %s créé\n", CHEMIN_MEMOIRE);

    /* - 2 - configure la taille du segment de mémoire partagée */
    ftruncate(descripteur_memoire, taille_memoire);

    /* - 3 - projette le segment de mémoire partagée dans l'espace mémoire du ↵
        processus */
    memoire = (struct memoire_t *) mmap(NULL,
        taille_memoire,
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        descripteur_memoire,
        0);
    if (memoire == MAP_FAILED) {
        traiter_erreur("Erreur lors de l'appel à mmap\n");
    }

    fprintf(stderr, "Mémoire de %d octets correctement allouée.\n", ↵
        taille_memoire);

    entier = 1;
    for (EVER) {
        /* - 4 - écriture dans le segment mémoire projeté en accédant aux champs↵
            : */
        memoire->valeur = entier;
        memoire->racine = sqrt(entier);
        sleep(5);
        entier++;
    }
}

```

```

    }

    /*
     * - 5 - ferme le segment de mémoire partagée - Jamais atteint d'où la ↵
     * gestion
     * par interception du signal SIGINT (<CTRL-C>).
     */
    if (shm_unlink(CHEMIN_MEMOIRE) != 0) {
        traiter_erreur("Erreur lors de l'appel à shm_unlink\n");
    }

    exit(EXIT_SUCCESS);
}

```

```

/**
 * @file consommateur_mem.c
 * Processus consommateur qui lit indéfiniment dans une mémoire partagée un ↵
 * entier
 * ainsi que sa racine carrée.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <signal.h>

#define EVER ;;
#define CHEMIN_MEMOIRE "/mem_racine" /* Voir /dev/shm sous Linux */

/** Structure utilisée dans le segment de mémoire partagée */
struct memoire_t {
    int valeur; /*!< valeur entière */
    double racine; /*!< racine carrée */
};

void traiter_erreur(char *message)
{
    fprintf(stderr, "%s", message);
    exit(EXIT_FAILURE);
}

/**
 * Traitant pour sortir de la boucle infinie et quitter en libérant la mé↵
 * moire.
 */
void traiter_sigint(int signum)

```



```

{
    if (shm_unlink(CHEMIN_MEMOIRE) < 0) {
        traiter_erreur("Erreur lors de l'appel à shm_unlink\n");
    }
    exit(EXIT_SUCCESS);
}

int main(int argc, char *argv[])
{
    int descripteur_memoire;
    struct memoire_t *memoire;
    int taille_memoire;

    taille_memoire = (1 * sizeof(struct memoire_t));

    signal(SIGINT, traiter_sigint);

    /* - 1 - ouvre le segment de mémoire partagée comme un fichier */
    descripteur_memoire = shm_open(CHEMIN_MEMOIRE, O_RDWR, S_IRUSR|S_IWUSR);
    if (descripteur_memoire < 0) {
        traiter_erreur("Erreur lors de l'appel à shm_open\n");
    }
    fprintf(stderr, "Objet de mémoire partagée %s ouvert", CHEMIN_MEMOIRE);

    /* - 2 - projette le segment de mémoire partagée dans l'espace mémoire du ←
    processus */
    memoire = (struct memoire_t *) mmap(NULL, ←
        taille_memoire,
        PROT_READ | PROT_WRITE,
        MAP_SHARED,
        descripteur_memoire,
        0);
    if (memoire == MAP_FAILED) {
        traiter_erreur("Erreur lors de l'appel à mmap\n");
    }
    fprintf(stderr, "Mémoire de %d octets correctement allouée.\n", ←
        taille_memoire);

    for (EVER) {
        printf("valeur : %d \n", memoire->valeur);
        printf("racine : %f \n", memoire->racine);
        sleep(3);
    }

    if (shm_unlink(CHEMIN_MEMOIRE) != 0) { /* Jamais atteint */
        traiter_erreur("Erreur lors de l'appel à shm_unlink\n");
    }

    exit(EXIT_SUCCESS);
}

```

}

Sémaphores POSIX

De la même manière que pour l'utilisation de segments de mémoire partagée, les sémaphores POSIX sont considérés comme des fichiers (cf. cours pour les propriétés).



Un programme utilisant des sémaphores POSIX doit en principe être lié avec l'option `-pthread` sous Linux (pas sur Mac OS X).

Voici un exemple de fonctions de manipulation d'un sémaphore POSIX.

```
/**
 * @file posix_semaphore.c
 *
 * Exemple de fonctions mettant en oeuvre les sémaphores POSIX.
 */

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include <semaphore.h>

#define NOM_SEMAPHORE "/sem_morpion"

typedef sem_t semaphore_t;

void traiter_erreur(char * message)
{
    fprintf(stderr, "%s", message);
    exit(EXIT_FAILURE);
}

/**
 * Créé un sémaphore POSIX de nom "NOM_SEMAPHORE". A faire par le processus
 * organisateur.
 */
semaphore_t * creer_et_ouvrir_semaphore()
{
    semaphore_t *sem;

    sem = sem_open(NOM_SEMAPHORE, O_CREAT|O_RDWR, S_IRUSR|S_IWUSR, 1);

    return sem;
}
```

```
/**
 * Ouverture du sémaphore POSIX de nom "NOM_SEMAPHORE". A faire par le
 * processus participant et doit exister.
 */
semaphore_t * ouvrir_semaphore()
{
    semaphore_t *sem;

    sem = sem_open(NOM_SEMAPHORE, O_RDWR, S_IRUSR|S_IWUSR, 0);

    return sem;
}

/**
 * Destruction du sémaphore.
 */
void detruire_semaphore(semaphore_t * sem)
{
    sem_close(sem);
    sem_unlink(NOM_SEMAPHORE);
}

/**
 * Opération P() - prise du sémaphore : à faire avant tout appel
 * à une ressource critique, ici un accès (lecture ou écriture)
 * à la mémoire partagée.
 */
void P(semaphore_t * sem)
{
    if (sem_wait(sem) < 0) {
        traiter_erreur("Erreur d'appel de l'opération P()\n");
    }
}

/**
 * Opération V() - libération du sémaphore : à faire après tout
 * appel à une ressource critique, ici un accès (lecture ou
 * écriture) à la mémoire partagée.
 */
void V(semaphore_t * sem)
{
    if (sem_post(sem) < 0) {
        traiter_erreur("Erreur d'appel de l'opération V()\n");
    }
}
```