LIFPCA – Programmation Concurrente et Administration Système

Fascicule de Travaux Pratiques

Les sujets de TPs ont été réalisés au fil des années, par les différents responsables de LIFPCA :
Fabien Ricco, Yves Caniou, Matthieu Moy, Grégoire Pichon, Florence Zara

Table des matières

TP 1 – Threads & Fractales **3**

TP 2 – Producteur-Consommateur & Fractales **6**

TP 3 – Mécanisme de synchronisation **8**

TP 4 – Ordonnancement **11**

TP 5 – Administration Système **13**

ANNEXE 1 – Accès à distance **20**

ANNEXE 2 – Accès aux machines virtuelles (VMs) **22**

Compétences

A l'issue des TPs, voici les compétences à acquérir :

Compétences en programmation et méthodologie

- ❑ Développer un programme multithreadé en C++ en employant les threads POSIX.
- ❑ Implémenter des mécanismes d'exclusion mutuelle.
- ❑ Résoudre des problèmes classiques de synchronisation (producteur-consommateur, lecteurs-rédacteurs, philosophes, etc.).
- ❑ Éviter les erreurs de synchronisation lors de la conception d'un algorithme concurrent.
- ❑ Déboguer un programme présentant des erreurs liées à la concurrence.
- ❑ Employer des algorithmes d'ordonnancement tels que FCFS, SJF, Priorités, Round Robin ou SRTF.
- ❑ Installer, mettre à jour et supprimer des logiciels à l'aide du gestionnaire de paquets.
- ❑ Configurer les permissions des fichiers et répertoires.
- ❑ Utiliser les pages de manuel (man) et la documentation système pour résoudre un problème.
- ❑ Travailler de manière autonome dans un environnement Linux.

TP 1 – Threads & Fractales

Note

Le premier exercice devrait être fait en 1h30.

Pour compiler : `g++ fichier.cpp -std=c++11 -pthread -o executable`

1 Premiers pas avec les threads

Vous devez paralléliser un petit programme : `lancement.cpp` disponible sur le site de l'UE. Ce dernier répète un certain nombre de fois un calcul qui n'a pas d'importance. Le but de l'exercice est de savoir si vous pouvez paralléliser les appels à la fonction `fct()`, récupérer le résultat (qui est le temps écoulé pendant le calcul d'après la fonction) et comparer ce temps avec le temps réellement écoulé.

Q1. Dans la fonction `main()` remplacez la boucle qui appelle `nbthreads` fois la fonction `fct()` par le lancement de `nbthreads` threads qui exécutent la fonction.

Q2. Si cela ne vous est pas déjà arrivé, faites une erreur d'argument entre le lancement du thread et la fonction, afin de vous habituer à lire les messages d'erreur de compilation. Par exemple, essayez un passage par référence en oubliant `std::ref()` au moment de l'appel : le message d'erreur est rarement convivial !

Q3. Normalement, dès qu'il y a un nombre de threads important, vous pouvez observer que les messages qu'ils affichent se mélangent. Sachant que les fonctions système sont prévues pour fonctionner en environnement multithread, pourquoi y a-t-il ce mélange et comment l'éviter ?

Q4. À la fin des threads obtenez le résultat de la fonction et affichez le temps que chaque thread pense avoir mis pour s'exécuter ainsi que la somme de ces valeurs.

La commande `time` permet de voir les différents temps utilisés par une commande :

```
$ time ./lancement 20
Th principal : lancement de 10 fois la fonction
0 lancement de la fonction pour 5 iterations
...
Th principal : Le temps total de calcul est 9.87181

real    0m9.876s
user    0m9.863s
sys     0m0.002s
```

- *real* est le temps écoulé pendant le calcul.
- *user* est la somme des temps processeur utilisés par les threads en mode utilisateur (pour faire les calculs eux-mêmes).
- *sys* est la somme des temps processeur passés en mode noyau (pour faire les opérations système comme les affichages).

Q5. Faites plusieurs essais avec 2, 3, 4, 8, 10, 20 threads.

a) Pourquoi le temps utilisateur est-il supérieur au temps réel ?

b) La somme des temps que vous calculez est-elle liée au temps réel ? Au temps utilisateur ? Au temps système ?

c) Est-elle identique, et s'il y a une différence pourquoi ?

Note

Pour la dernière question il faut comparer cela avec le nombre de processeurs/cores de l'ordinateur (`cat /proc/cpuinfo`).

A retenir

Voici ce qu'il faut bien retenir :

- option de compilation pour un code multi threadé ;
- création d'un thread qui va exécuter une fonction ;
- gestion des threads en utilisant un `std::vector<thread> tab_threads` ;
- gestion de la terminaison de chacun des threads lancés ;
- utilisation d'un tableau pour récupérer facilement le résultat de chacun des threads.

2 Calcul de fractales

2.1 Prise en main du code initial

Nous allons paralléliser le calcul d'une image fractale en découpant le travail (calcul de toute l'image) en petites tâches (calcul d'une tranche de l'image). Les threads lancés se partageront ainsi ces tâches. Le calcul de la fractale consiste à évaluer une fonction permettant de déterminer la couleur en chacun des pixels de l'image. Cette fonction mathématique n'est pas à considérer, elle est déjà écrite pour vous. Afin d'accélérer le calcul, nous vous demandons donc de faire réaliser ce travail par plusieurs threads. Chaque thread faisant un certain nombre de tranches d'écran.

- Téléchargez et décompressez l'archive `mandel.zip` se trouvant sur la page web de l'UE. Compilez le programme (`make`) puis lancez-le (`./mandel`).
  Vous verrez apparaître une fenêtre graphique sur laquelle se dessine progressivement la fractale de Mandelbrot. Pour l'instant, l'image est divisée en 10 tranches, et un seul thread calcule ces tranches, les unes après les autres (on peut voir à l'œil nu qu'il n'y a jamais plus d'une tranche en cours de calcul).
- Vous pouvez faire varier le nombre de tranches d'écran avec par exemple : `./mandel --nb-slices 100`.
  Les fichiers `display.*` et `mandel.*` gèrent respectivement l'affichage à l'écran et le calcul de la couleur de chaque pixel de la fractale. Vous n'aurez pas besoin de modifier ces fichiers.
- **Le fichier `main.cpp` est celui qui nous intéresse** : ouvrez-le et regardez son code, en particulier la fonction `main()` et `draw_screen_sequential()` qu'elle appelle par défaut.
  Si le calcul est trop rapide, vous pouvez le ralentir artificiellement avec `--slow` (répétez l'option plusieurs fois pour ralentir encore). Pour afficher plus de traces de debug, utilisez `--verbose`.
  Pour le plaisir des yeux, vous pouvez jouer avec les options `--loop`, `--resize` (zoomer sur la position du pointeur de la souris à chaque itération).

2.2 Principe de la parallélisation avec liste de tâches

Le temps nécessaire au calcul de chaque pixel n'est pas constant car il dépend des coordonnées du pixel considéré. *Il n'est donc pas efficace de donner le même nombre de tranches à chaque thread.*

Pour faire le calcul, vous devez **créer dès le départ un nombre fixé de threads** de calcul et une **liste des tâches à effectuer**. Chaque thread va devoir chercher dans la liste une tâche à faire, effectuer cette tâche et recommencer jusqu'à ce que la liste soit vide. La liste est un objet partagé et contient uniquement le numéro des tranches à traiter. Vous devrez faire bien attention à ce que toutes les tranches soient traitées une et une seule fois. Le code préparé contient plusieurs tests dont le but est de vérifier cela. Le principe de

la gestion de la liste de tâches vous est fourni dans la fonction `draw_screen_worker()`, qui va en boucle chercher du travail (`get_slice()`) puis effectuer la tâche correspondante (`compute_and_draw_slice()`), jusqu'à ce que `get_slice()` renvoie -1.

- Lisez le code de `draw_screen_worker()`. Essayez de remplacer temporairement l'appel à `draw_screen_sequential()` dans `main()` par `draw_screen_worker()` : le résultat est le même. Avec un seul travailleur, le calcul reste séquentiel. Après cette vérification, restaurez le code original avec un appel à `draw_screen_sequential()`.
- Le but de la suite du TP sera de lancer plusieurs travailleurs en parallèle, chacun exécutant la fonction `draw_screen_worker()`, tous les threads partageant la liste de tâches à exécuter.

  Lancez par exemple `./mandel --nb-threads 2`.

  Le programme s'arrête avec le message `draw_screen_thread not yet implemented`.

  Cela va être l'objectif de la suite du TP !

2.3 Implémentation à faire

Vous devez maintenant implémenter la fonction `draw_screen_thread()` pour suivre le schéma décrit dans le paragraphe précédent.

Q1. Modifier la fonction pour lancer `number_of_threads` threads, chacun exécutant `draw_screen_worker`

Q2. Regardez le mécanisme proposé dans le code pour distribuer les tranches d'écran aux threads : il s'agit de la fonction `get_slice()` qui utilise la variable globale, donc partagée, `last_slice` (dernière tranche calculée).

Q3. Faire en sorte que les threads de calcul traitent les tranches jusqu'à ce qu'il ne reste plus de tranche à calculer (fonction `get_slice()` est prévue pour cela et retourne -1 lorsque la liste est vide). Attention, la fonction `get_slice()` n'est pas prévue pour fonctionner en environnement multithread (elle n'est pas *thread safe*, et a même été écrite volontairement pour poser un maximum de problèmes en cas d'accès concurrent). Vous devez faire en sorte que le programme ne calcule pas plusieurs fois la même tranche, et que toute ces tranches soient bien calculées (si ce n'est pas le cas, cela est visible sur l'image obtenue).

Q4. À ce stade, le calcul est complet et doit afficher la fractale correctement. Modifiez-le maintenant pour que le thread principal obtienne le nombre de tranches calculées par chaque thread, faire la somme et vérifier que ce nombre correspond à celui des tranches de l'image (la vérification est faite pour vous dans `main()` à l'appel de `draw_screen_thread()`).

Q5. Nous vous avons fourni une fonction `draw_rect_thread_safe()` qui s'occupe de l'affichage d'un rectangle de l'écran de manière *thread-safe*, c'est à dire qui ne pose pas de problème lors d'accès concurrents. Essayez de remplacer cet appel par `draw_rect()`. Vous devriez obtenir un problème d'accès concurrent à l'affichage (la bibliothèque X11 va afficher une erreur). Cette erreur ne se produit pas sur certains systèmes, donc ne vous acharnez pas si vous ne l'avez pas. À vous de faire en sorte qu'il n'y ait plus d'accès concurrents à l'affichage (sans utiliser `draw_rect_thread_safe()`, qui est en fait le corrigé de cet exercice).

Q6. Vérifiez expérimentalement que le calcul est plus rapide avec plus d'un thread (`--nb-threads ...`). Comparez le nombre de threads optimal avec le nombre de cœurs de la machine (`/proc/cpuinfo`). Le nombre de tranches (`--nb-slices ...`) peut avoir une influence (avec trop peu de tranches, certains threads n'auront plus rien à faire en attendant que la dernière tranche soit calculée, avec trop de tranches on augmente le nombre d'appels à `get_slice()`).

TP 2 – Producteur-Consommateur & Fractales

Le but de ce TP est de modifier le code du TP1 concernant le calcul de fractale de Mandelbrot en parallèle afin de **mettre en place un système de producteur-consommateur**.

Note

- Si vous avez bien réussi le TP1, vous pouvez continuer en utilisant votre code.
- Sinon, commencez par récupérer l'archive `mandel-threads.zip` sur la page de l'UE.
- Compilez et exécutez le code et vérifiez qu'il s'agit bien du travail demandé au TP1.

Au TP1, nous avions utilisé un **mutex** pour empêcher les accès concurrents à l'affichage.

Dans ce TP2, nous allons **dédier un thread à l'affichage** (le thread s'occupera d'appeler `draw_rect()`). Ce thread sera le seul à y accéder, donc il n'y aura plus de problème d'exclusion mutuelle.

Les threads de calcul (les `draw_screen_worker` du TP1) n'appelleront plus `draw_rect()`, mais ils enverront le rectangle d'écran à afficher au thread chargé de l'affichage.

La communication entre les threads de calcul et le thread d'affichage se fera au moyen d'un **schema « producteur-consommateur »**, comme vu en TD. Il s'agit d'une classe C++ (un moniteur), qui expose les méthodes suivantes :

- `void put(element)` : appelée par le ou les producteurs pour envoyer un élément au consommateur. Cette fonction est bloquante si la file est pleine.
- `element get()` : appelée par le ou les consommateurs pour récupérer un élément envoyé par un producteur. Cette fonction est bloquante si la file est vide.

Les éléments échangés sont des instances de la structure suivante qui décrit un rectangle à afficher :

```
struct rect {
        int slice_number;
        int y_start;
        rect(int sn, int y) : slice_number(sn), y_start(y) {};
};
```

Les threads de calculs enverront les rectangles avec :

```
prod_cons.put(rect(slice_number, y));
```

Le thread chargé de l'affichage récupérera ces valeurs pour appeler `draw_rect()` dessus avec :

```
rect r = prod_cons.get();
```

1 Une classe pour le producteur-consommateur

Q1. Dans de nouveaux fichiers C++ (`ProdCons.h` et `.cpp`), créez une classe `ProdCons` contenant les deux méthodes `put()` et `get()` décrites ci-dessus. Si vous êtes à l'aise en C++, écrivez une classe template pour que les arguments de `put()` et `get()` soient génériques.

Q2. Ajoutez les données nécessaires pour implémenter une file d'attente (FIFO) dans la classe. En TD, nous avons codé un buffer circulaire à la main avec un tableau ; vous pouvez aussi utiliser une structure toute faite de C++ comme `std::list` ou `std::queue` (vous aurez besoin des méthodes `push()`, `pop()`, `front()` et `size()`).

Q3. Ajoutez le nécessaire pour que cette classe puisse agir comme un moniteur de Hoare.

Q4. Donnez le corps des fonctions `put()` et `get()`.

2 Emploi de la classe pour le producteur-consommateur

Utilisez maintenant votre classe dans le code de calcul de Mandelbrot. Notez que plusieurs noms de fonctions ne refléteront plus exactement ce que les fonctions en question effectuent...

Q1. Instanciez votre classe `ProdCons`. Le plus propre est de l'instancier dans `draw_screen_thread()`, mais vous pouvez aussi le faire en variable globale pour simplifier.

Q2. Écrivez la fonction `x_thread_function()` qui sera exécutée par le thread chargé de l'affichage. Cette fonction doit avoir accès à l'instance de `ProdCons`. Son rôle se limitera à récupérer les éléments contenus dans le `ProdCons`, puis à les afficher.

Q3. Modifier la fonction `draw_screen_worker()` qui sera exécutée par les threads de calculs pour envoyer un rectangle d'écran à afficher au thread d'affichage. Cette fonction doit également avoir accès à l'instance de `ProdCons`.

Q4. Dans la fonction `draw_screen_thread()`, lancez le thread d'affichage en début de calcul. À quel moment pouvez-vous faire un `join()` sur le thread ? Immédiatement ? Plus tard ? Pourquoi ?

Q5. Modifiez la fonction `compute_and_draw_slice()` pour lui faire envoyer les rectangles au thread chargé de l'affichage.

Q6. Une problématique intéressante est celle de la terminaison : comment la fonction `x_thread_function()` est-elle en mesure de déterminer qu'elle a affiché la totalité de la fractale ? Modifier votre code en conséquence, si vous ne l'aviez pas pris en compte.

Q7. Testez l'ensemble de votre code !

Q8. Notre classe `ProdCons` peut aussi être utilisée pour gérer la liste des tâches (la fonction `get_slice()` que nous avons utilisée lors du TP1).
On peut alors instancier un thread chargé d'appeler `get_slice()` et d'envoyer les tranches d'écrans à calculer aux threads de calcul. De cette manière, le thread producteur sera le seul à appeler `get_slice()` (on pourra alors supprimer le mutex qui protégeait son accès).

Si le temps le permet, mettez en place ce producteur-consommateur.

3 Un peu de recul

Cette section est destinée à vous faire réfléchir à ce que vous venez de coder. Son objectif n'est pas pratique, mais réflexif. Comprendre pourquoi vous utilisez un outil plutôt qu'un autre est essentiel. Il n'est pas obligatoire de répondre à ces questions pendant le TP, mais nous vous recommandons de leur consacrer un peu de votre temps à un moment ou un autre.

- Pourquoi utilise-t-on une structure `rect` ? Pourquoi ne pas simplement utiliser deux producteurs-consommateurs d'`int` ? Interrogez-vous sur l'utilité d'une structure de données, l'utilité d'une classe.
- Cette version du producteur-consommateur place les producteurs et les consommateurs en exclusion mutuelle. Il est impossible de consommer en même tant qu'une production, et vice-versa. Cette restriction est-elle inévitable ? Après-tout, les premières valeurs dans la file du producteur-consommateur ne vont pas changer en raison d'une insertion, pour peu qu'on utilise une structure de données qui supporte l'insertion sans déplacer toutes ses valeurs (une liste chaînée par exemple). Pouvez-vous ébaucher une variante de `ProdCons` dans laquelle il est possible de lire une valeur tout en produisant une autre valeur, du moment qu'il y a au moins une valeur de disponible dans le `ProdCons` ?

TP 3 – Mécanisme de synchronisation

Note

Finir les TP1 et TP2 avant d'attaquer le TP3. Il faut avoir compris

- l'utilisation des threads avec les mutex ;
- la gestion de l'exclusion mutuelle pour les sections critiques ;
- la mise en place d'un mécanisme de type producteur-consommateur ;
- l'implémentation d'un moniteur de Hoare.

Dans ce TP, nous allons voir la mise en place de synchronicité dans l'exécution de tâches.

1 Synchronisation simple

Le but ici est d'écrire un programme qui effectue les tâches A_1 et A_2 en parallèle, et après la terminaison de ces deux taches, la tâche B. Les tâches A_1 et A_2 exécutent du code arbitraire, par exemple Fibonacci() sur un nombre tiré aléatoirement (ou tout simplement `sleep`;D). La tâche B attend que les tâches A_1 et A_2 soient terminées avant de s'exécuter. Attention, les trois tâches doivent s'exécuter en parallèle : vous ne pouvez pas simplement lancer A_1 et A_2 en parallèle, puis faire un `join` avant de lancer B ! Voici le squelette du `main` de votre programme :

```
int main() {
  for(...) {
    std::thread A_1(...);
    std::thread A_2(...);
    std::thread B(...);
    A_1.join(); A_2.join(); B.join();
    }
  return 0;
}
```

Q1. Proposez une solution et discutez-en avec votre encadrant. Votre solution est-elle facilement généralisable au cas n threads ?

Q2. À quel type de problème cela correspond-il ?

2 Barrière de synchronisation

Nous vous proposons ici de coder une **barrière de synchronisation** afin de faire fonctionner un code d'exemple. Le code principal de votre `main()`, doit demander à l'utilisateur le nombre de threads qui s'exécuteront, ainsi qu'une valeur `nbtours` qui représente le nombre d'itérations d'une boucle `for`. Vous pouvez demander la valeur de `nbtours` interactivement via `cin` ou `scanf`, ou en ligne de commande en utilisant les paramètres `argc` et `argv` du `main()`.

Le code principal lance ensuite les threads qui exécuteront le pseudo-algorithme suivant :

```
// Chaque thread execute nbtours fois la sequence :
    Tache()
    Barriere() // objectif du TP
```

Le pseudo-algorithme de $Tache()$ est le suivant :

```
Affiche l'heure
Tire un nombre X aleatoirement dans U[10,20]
Calcul de Fibonacci(X)
Calcul de la duree passee pendant le calcul et l'affiche
```

Votre travail consiste surtout à coder l'opération `Barriere()`. Vous êtes libre de procéder comme vous le souhaitez, à ceci près que vous n'avez pas le droit d'utiliser la structure `pthread_barrier_t`, ni les classes `std::latch` et `std::barrier` de C++20 (qui sont les classes que nous cherchons à ré-écrire).

Note

Si vous le souhaitez, vous pouvez arrêter la lecture du sujet ici afin de coder votre solution sans aide supplémentaire (sachez tout de même qu'une difficulté est qu'il peut y avoir plusieurs appels à Barriere sur le même objet du fait de la boucle autour de cet appel. Il est conseillé de démarrer par une version simplifiée qui ne gère qu'un seul appel).

2.1 Barrière non-réutilisable

Dans un premier temps, nous supposerons qu'il y a N threads (N étant une constante à définir dans le code) et que chaque thread n'appelle `wait()` qu'une seule fois. Nous appellerons notre classe `latch` car c'est ce que fait la classe `std::latch` de C++20.

Pour réaliser ce code, nous utiliserons un compteur initialisé à 0 comptant le nombre de threads ayant atteint la barrière (i.e. ayant démarré l'appel à `wait()`). Les threads seront tous débloqués quand ce compteur arrive à N.

Q1. Écrivez le squelette de classe `latch` comprenant une méthode `wait()` dont le corps restera vide.

Q2. Écrivez le programme principal `main()` instanciant N threads. Chacun des threads :
1) affiche la chaîne `"avant\n"`
2) accède à la barrière
3) affiche la chaîne `"apres\n"`, puis termine son exécution.

Q3. Que devrait afficher l'exécution de la fonction `main()` écrite pour $N = 3$? Vous pourrez vérifier votre prédiction juste après la question suivante.

Q4. Complétez l'implémentation de la classe `latch` en ajoutant les champs privés, le corps de la fonction `wait()`, et celui du constructeur de la classe `latch`.

2.2 Barrière réutilisable

Nous allons maintenant écrire une barrière plus générique, où la fonction `wait()` peut être appelée plusieurs fois d'affilée par chaque thread. Par exemple, chaque thread peut exécuter le code suivant :

```
1 void barrier_infinite(barrier &b) {
2     while (true) {
3         b.wait();
4    }
5 }
```

La difficulté est qu'en débloquant les threads (quand N threads ont appelé `wait()`), on peut arriver dans une situation où certains threads démarrent leur second appel à `wait()` alors que d'autres n'ont pas terminé leur premier appel. La solution que nous allons utiliser ici est d'avoir un compteur de génération :

- Initialement, le compteur de génération vaut 0.
- Quand un thread fait un appel à `wait()` pendant la génération i, il est bloqué jusqu'à ce que le compteur de génération devienne différent de i.
- Quand un thread fait le Nième appel à `wait()` de la génération i, il incrémente i pour débloquer tous les threads (et lui-même n'est pas bloqué).

Q1. Proposez une nouvelle implémentation de la classe barrière que nous nommerons `barrier` puisqu'elle correspond désormais à la classe `std::barrier` de C++20.

Q2. Votre programme principal instanciera N threads exécutants chacun le code suivant :

```
1 void barrier_ngen(barrier &b) {
2         for (int i = 0; i < 5; ++i) {
3                 sleep(1); // attente d'une seconde
4                 b.wait();
5         }
6 }
```

On néglige les temps de calcul et on considère que seule la fonction `sleep(1)` prend du temps, sachant que `sleep()` fait une attente passive, c'est-à-dire que d'autres threads peuvent s'exécuter pendant son exécution.

Q3. Combien de temps mettra le programme à s'exécuter avec $N = 10$?

Q4. Que se passe-t-il si on exécute le programme sur un système mono-cœur ?

Pour ces deux questions, faites d'abord une prédiction, par exemple en dessinant un chronogramme qui montre l'exécution du programme, puis vérifiez votre prédiction expérimentalement.

TP 4 – Ordonnancement

1 Introduction

Pour ce TP, nous allons jouer avec les politiques d'ordonnancement temps-réel du noyau Linux (`SCHED_FIFO`, `SCHED_RR`). Activer ces politiques est potentiellement dangereux pour le système (une boucle infinie en priorité temps-réel peut figer l'ensemble du système), donc interdit pour des simples utilisateurs : vous aurez besoin de passer root pour le faire. **Vous ne pourrez donc pas faire ce TP sur les machines physiques de l'université.**

Nous allons utiliser l'infrastructure de *cloud computing* du département financée par la région Rhône-Alpes. C'est un ensemble de machines pilotées par le logiciel `OpenStack` avec lequel vous allez avoir un premier contact. Vous allez l'utiliser afin de créer une machine virtuelle, que vous pourrez conserver, effacer (mais perdre ainsi toutes les configurations effectuées), re-générer... La machine virtuelle (VM) créée va ici nous servir de base d'expérimentation, et dans le TP suivant d'entraînement à des manipulations d'administration systèmes que vous pourrez ensuite effectuer sur vos machines personnelles.

Note

Référez-vous aux consignes d'accès à distance ou d'accès / création à une VM (cf. la page Web), si vous ne vous rappelez plus comment faire. Il faudra vous connecter à la VM, savoir récupérer un fichier (code C++) se trouvant sur la page Web, le compiler, l'exécuter en tant que `root`.

2 Modification de l'ordonnanceur

Dans cet exercice, vous allez utiliser explicitement l'ordonnanceur du noyau Linux. C'est assez dangereux, car un processus prioritaire qui ne s'arrête pas, par définition, bloque l'ordinateur. Vous devez donc **utiliser les machines virtuelles**.

Le code `sched.cpp` a été préparé (il est sur la page Web), il lance un certain nombre de threads qui font des calculs en affichant de temps en temps des informations. Pour le moment, le code de la fonction `change_ordonnancement()` n'est pas fait et l'ordonnancement n'est pas modifié.

Q1. Complétez le code de la fonction `change_ordonnancement()` pour qu'elle change l'ordonnancement du thread qui l'appelle. Il n'y a pas de fonctions C++ définies dans la bibliothèque standard pour manipuler l'ordonnanceur. Vous allez devoir utiliser les fonctions C suivantes pour faire ce travail :

- Retourner le numéro du thread qui l'appelle :

```
pthread_self();
```

- Récupérer les paramètres d'ordonnancement courant (elle vous permettra d'initialiser les variables) :

```
int pthread_getschedparam(pthread_t target_thread,
                          int *politique,
                          struct sched_param *param);
```

- Modifier la politique d'ordonnancement et ses paramètres :

```
int pthread_setschedparam(pthread_t target_thread,
                          int politique,
                          const struct sched_param *param);
```

- Vérifiez la valeur de retour de la fonction (notamment pour vous assurer que vous avez bien les droits pour changer la politique et la priorité, il faut être `root` pour le faire ; elle est de 0 si tout est correcte). La valeur de la politique est définie dans des macros : `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER`.

Q2. Lancez 3 threads RR (`Round Robin`), l'un de priorité 4 et deux autres de priorité 2.

Q3. Lancez 3 threads FIFO de priorité identique.

Q4. Lancez 1 threads FIFO et 2 RR de priorité identique.

Q5. Lancez 1 thread FIFO de priorité 99 et 2 autres FIFO de priorité plus faibles.

Q6. Pouvez-vous stopper le programme pendant que des threads très prioritaires tournent ? Pourquoi ?

Q7. Si vous avez votre propre ordinateur, refaites les expériences sur ce dernier. Avez-vous les mêmes résultats ? Pourquoi ?

3 Intégration par la méthode des rectangles

Les sommes de Riemann sont des sommes approximant des intégrales. Ainsi, si on considère $f : [a, b] \to \mathbb{R}$ une fonction partout définie sur le segment $[a, b]$; un entier $n > 0$ et une subdivision *régulière* définie pour $0 \leq k \leq n$ par :

$$x_k = a + k\frac{b-a}{n}.$$

Alors la somme de Riemann est définie comme

$$S_n = \frac{b-a}{n} \sum_{k=1}^{n} f(a + k\frac{b-a}{n}),$$

c'est-à-dire

$$S_n = \sum_{k=1}^{n} (x_k - x_{k-1}) f(x_k).$$

Plus la valeur de n est grande, plus cette méthode des rectangles pour le calcul des intégrales donne une estimation proche de la valeur cherchée de l'intégrale.

Nous avons accès à des machines multi-cœur, et voulons en faire bon usage pour calculer des résultats d'intégration.

Q1. Proposez un programme retournant la valeur de la somme totale calculée et qui sera capable de s'adapter au nombre de cœurs fourni par l'utilisateur en ligne de commande (de sorte que chacun soit utilisé pour calculer une partie d'intégrale).
Vous pourrez utiliser des fonctions à intégrer plus ou moins complexes pour vous assurer de la validité de votre code, et pour les questions suivantes.

Q2. Donnez 2 moyens de savoir combien de cœurs dispose la machine que vous utilisez actuellement.

Q3. À l'aide de la fonction `gettimeofday()`, mesurez la durée du programme pour un nombre de cœurs valant 1, 2, 4, 8 et 64. Commentez les résultats obtenus !

TP 5 – Administration Système

Note

En préparation de ce TP, quelques pages Web à lire qui abordent les principales informations théoriques à retenir :

- `https://www.debian.org/doc/manuals/debian-faq/pkg-basics.fr.html`
- `https://www.debian.org/doc/manuals/debian-faq/pkgtools.fr.html`
- `https://www.debian.org/doc/manuals/debian-faq/uptodate.fr.html`

Il est fortement conseillé d'avoir également fait les DMs présents sur la page Web.

1 Administration Système

Note

Les commandes à maîtriser : `top source alias less tail head ps grep whoami chown chgrp passwd adduser addgroup sudo su lsmod modprobe apt-get aptitude dpkg man`, ainsi que certaines de leurs options.

1.1 Création/re-création de VM

Pour ce TP, vous devez utiliser une machine virtuelle sur laquelle vous pouvez être administrateur. On supposera par la suite que `IP_VM` est son IP. Grâce au DM0, vous savez comment configurer un environnement agréable pour travailler, que ce soit sur votre compte Lyon 1 ou sur votre VM (notamment avec des alias comme `l='ls --color -l'`, un prompt `PS1` défini correctement et en couleur, etc.).

1.2 Configurations d'environnement

Vous pouvez remarquer que la complétion existe bien sur votre compte Lyon1, mais pas la complétion "intelligente". Nous avons vu dans le DM0 qu'elle était mise en place automatiquement grâce à la commande `source /etc/bash_completion` sourcée au démarrage d'un nouveau shell. Mais ce fichier n'existe pas sur les installations Fedora.

Q1. À partir de la VM, comment connaître son adresse IP ?

Q2. Le fichier `/etc/bash_completion` est-il le seul intéressant ? Pour le savoir, regardons quel package l'a installé (à moins que vous vous souveniez de son nom ?) : tapez `dpkg -S /etc/bash_completion`. Cela permet justement de donner le package qui a installé le fichier donné.

Q3. Comment lister tous les fichiers installés par le package contenant `/etc/bash_completion` ?

Q4. On voit que le package installe notamment `/etc/profile.d/bash_completion.sh`. Qu'est-ce que ce fichier, et que contient-il (et pourquoi) ? Quelle différence avec `/etc/bash_completion` ?

Q5. On voit également le fichier `/usr/bin/dh_bash-completion`. Si on regarde son contenu, peut-on savoir à quoi il sert à priori ?

Q6. À l'aide de la commande `rsync` (ou `scp`), copiez le fichier `/etc/bash_completion` de votre VM sur votre compte Lyon1, par exemple dans `~/.bash_completion_VM`.

Q7. Sourcez-le. Est-ce que cela change quelque chose à l'environnement ? *A*ny comments ?

1.3 Retour sur les alias et fonctions

Q1. Toujours taper `ls -l --color` est long. Proposez un alias *l* pour faire la même chose, et rendez-le persistant !

Q2. La fonction suivante utilise le package `adb` pour communiquer avec un téléphone android. Que fait-elle ?

```
# $1 must be provide as date like 201509, 2015, or 20150926
function down_pics_from_phone {
    local photo_dir="/storage/sdcard1/DCIM/Camera"
    if [ "$1" == "" ] ;then
        adb shell "ls $photo_dir" > /dev/shm/list_pics2
        tr -d '\r' < /dev/shm/list_pics2 > /dev/shm/list_pics
        rm /dev/shm/list_pics2
        cat /dev/shm/list_pics
        echo List available in /dev/shm/list_pics
        echo Use $0 YYYYMMDD or subset to pull photos
        return
    fi

    adb shell "ls $photo_dir" | grep $1 > list_pics
    tr -d '\r' < list_pics > list_pics2
    for i in `cat list_pics2` ; do
        echo "Getting $i"
        adb pull $photo_dir/$i
    done
    rm list_pics list_pics2
}
```

1.4 Les utilisateurs et root – rappels de LIFSE

Q1. Avant tout, connectez-vous (ssh) sur la VM OpenStack en tant qu'utilisateur `etudiant` (reprenez l'énoncé du TP1 si besoin).

Q2. Quels sont tous les utilisateurs présents sur la machine ?

Q3. Sur votre VM, créez un autre utilisateur de login `lurong` (nom complet : Gai Luron), et donnez-lui un mot de passe différent de celui pour `etudiant` (rappel : la commande `adduser` est plus pratique que `useradd`, c'est en fait un wrapper.).

Q4. Depuis un shell appartenant à l'utilisateur `etudiant`, ouvrez un shell en tant qu'utilisateur `lurong`. Fermez ce shell pour revenir au shell de `etudiant`.

Q5. Depuis votre PC physique, ouvrez un nouveau terminal (pour conserver le shell `etudiant` ouvert), et dans ce nouveau terminal ouvrez une connexion sur votre VM en vous connectant *directement* en tant qu'utilisateur `lurong`.

Q6. Regardez le contenu du fichier `/etc/passwd` (il est accessible en lecture pour tout le monde, vous pouvez voir son contenu avec une commande comme `less` ou `cat`, ou l'ouvrir dans votre éditeur de texte). Vous devriez trouver une ligne pour l'utilisateur `lurong` contenant entre autres le nom complet que vous avez entré à la création, et d'autres informations qui ont été ajoutées automatiquement

comme le répertoire personnel `/home/lurong` et le shell par défaut `/bin/bash`. Notez que le mot de passe ne se trouve pas dans ce fichier (il y a un `x` dans la colonne où il aurait pu se trouver, qui signifie qu'il est stocké ailleurs).

Q7. Regardez maintenant le contenu du fichier `/etc/shadow`. Celui-ci n'est accessible que par `root`. Dans ce fichier, il y a bien une information sur le mot de passe, mais pas le mot de passe lui-même. Pourquoi ?

Q8. Quel est l'`UID` (User IDentifier, le nombre qui identifie l'utilisateur de manière unique) de l'utilisateur `lurong` ?

Q9. Comment connaître les groupes auxquels il appartient ?

1.5 Utilisateurs et gestion des droits

Q1. Arrangez-vous pour avoir deux terminaux ouverts, l'un avec un shell appartenant à `etudiant`, l'autre avec un shell `lurong`. Si vous avez bien suivi les consignes ci-dessus, vous l'avez déjà fait. Vous devriez aussi avoir des invites de commandes différentes dans ces deux terminaux.

Q2. Dans les deux terminaux, vérifiez que vous êtes bien celui que vous pensez à l'aide de la commande `whoami`.

Q3. Vérifiez que chaque utilisateur se trouve, pour l'instant, dans son propre répertoire personnel (c.-à-d. son répertoire de connexion. Rappel, on peut connaître le répertoire courant avec la commande `pwd`, et `cd` seule permet d'aller dans le répertoire de connexion), et que le répertoire de connexion de `etudiant` possède bien les droits `rx` pour `other` (c'est le cas par défaut sur beaucoup de distributions, mais peut ne pas l'être) – les changer si ce n'est pas le cas.

Q4. Dans le shell sur la VM, sous l'identité `etudiant`, créez un fichier avec la commande `touch poi` et exécutez `ls -l`. Modifiez ce fichier avec la commande de votre choix.

Q5. Dans le shell sous l'identité `lurong`, retrouvez le fichier `poi`. Faites un `ls -l` dessus et regardez son contenu. Essayez de le modifier. Que se passe-t-il ?

Q6. Revenez sous le shell de `etudiant` et exécutez la commande `chmod go+w poi`. Ré-essayez de modifier le fichier en tant que `lurong` (cela devrait marcher maintenant). Faites un `chmod go-w poi` en tant que `etudiant`, et vérifiez que la commande a bien coupé les droits en écriture.

Q7. Exécutez `chmod go-r poi` en tant que `etudiant` et vérifiez que les droits en lecture ont été coupés. Exécutez `chmod 644 poi` en tant que `etudiant` pour revenir à la situation précédente.

Q8. En tant que `lurong`, exécutez `chmod go+rw poi`. Vous devriez obtenir une erreur : vous n'êtes pas propriétaire du fichier, vous n'avez pas le droit de modifier ses permissions.

Q9. Ajoutez `lurong` au group `etudiant` à l'aide de la commande `usermod`, et vérifiez.

Q10. Est-ce que maintenant `lurong` peut modifier le fichier `poi` ?

Q11. Dans une console sur la VM, sous l'identité `root`, faites un `chmod go-rwx /etc/passwd`. En tant que `etudiant`, exécutez maintenant `ls -l` et expliquez ce que vous voyez (et pas ce que vous croyez voir). Essayez `ls -l ~`. Remettez les bons droits au fichier `/etc/passwd`.

Q12. En local (la question n'a pas de sens sinon), comment passer en mode console ? Revenir au mode graphique ?

1.6 Les packages et leur gestion

Q1. Quelle est la liste complète des packages installés ?

Q2. Que dois-je faire si je souhaite trouver un package contenant le mot clé `keyring` par exemple ?

Q3. Faire une mise à jour de la base des packages.

Q4. Démarrer une mise à jour du système (NE PAS METTRE À JOUR en annulant la mise à jour en répondant « n » à la dernière question. Mais regardez les informations affichées).

1.7 Serveur Nginx

Nous allons maintenant installer un serveur web sur votre VM. Un serveur web très connu est apache2, mais nous allons jouer dans un premier temps avec Nginx (à prononcer « engine-X »).

Q1. Comment savoir si Nginx est installé ?

Q2. Quel est le package concernant nginx ?

Q3. Voir si nginx est installé.
Quelle version serait installée si on décidait d'installer nginx ?

Q4. Installer le package `nginx`

Q5. Quels sont les fichiers installés par nginx ?

Q6. Voir les fichiers de configuration de nginx.

Q7. Où sont les fichiers de log de nginx

Q8. À votre avis, votre VM a t-elle un serveur ssh installé ? Comment le savoir ? Où sont ses fichiers ?

Q9. Vous pouvez regarder les packages `adb` (voir la question sur les fonctions), `french-conjugator`, `anki`, `nmap`, etc.

Q10. Installez le package `zmap` avec la commande `apt-get install zmap` (c'est un petit package, et ce n'est pas son installation ici qui est importante). Contrairement à `apt`, `apt-get` ne fait pas le ménage des packages téléchargés : on peut donc trouver les archives `.deb` dans `/var/cache/apt/archives/`. Copiez le package debian qui s'y trouve dans le répertoire de connexion, et explosez l'archive.

Q11. Dans un sous-répertoire `tmp`, explosez l'archive `control.tar.gz`. Que contient-elle ? À quoi servent ces fichiers ?

Q12. Dans un sous-répertoire `pmp`, explosez `data.tar.xz` et commentez.

1.8 Gestion des services avec systemd

Nginx est un logiciel prévu pour tourner en tâche de fond sur la machine. Il est lancé au démarrage et tourne en permanence même si personne n'est connecté à la machine. On appelle cela un *démon* (daemon en anglais). Son rôle est de répondre aux requêtes HTTP reçues, donc c'est aussi un *serveur*. Ce serveur fournit un *service* aux clients.

Q1. Vérifiez que votre machine répond bien quand un navigateur web l'interroge sur le port 80. Plusieurs solutions :

- Lancer votre navigateur web habituel sur `http://IP_VM/` (cela ne marchera que si vous avez un accès direct à la VM, donc pas si vous tentez une connexion depuis l'extérieur de l'université ni depuis le wifi).
- Depuis un shell qui tourne sur votre VM, lancez la commande `links http://localhost/` (si besoin, installez `links` au préalable). Links est un navigateur en mode texte, peu convivial mais cela présente entre autres l'avantage de pouvoir être lancé facilement via une connexion SSH.
- Si vous voulez vous amuser : faire un tunnel SSH pour accéder à `IP_VM:80` depuis votre PC physique.

Vous devriez voir apparaître la page d'accueil par défaut de nginx (« Welcome to nginx ! »). Si vous le voulez, vous pouvez aussi voir et modifier le contenu de cette page dans le fichier `/var/www/html/index.nginx-debian.html`.

- Interrogez systemd pour avoir le statut du service nginx avec la commande :

```
systemctl status nginx.service
```

Q2. Vérifiez « à la main » que le processus nginx tourne : `ps aux | grep nginx`

Q3. Coupez le service nginx avec la commande :

```
sudo systemctl stop nginx.service
```

Q4. Ré-essayez de charger la page web : vous aurez une erreur du type « connection refused ».

Q5. Interrogez systemd pour avoir le status du service nginx avec la même commande que ci-dessus. La ligne « Active » doit être passée de « active (running) » à « inactive (dead) ».

Q6. Vérifiez « à la main » que le processus nginx ne tourne plus : `ps aux | grep nginx`

Q7. Redémarrez le service :

```
sudo systemctl start nginx.service
```

Q8. Regardez à quoi ressemble le fichier de description du service pour systemd : `/etc/systemd/system/multi-user.t`
Vous n'avez pas besoin de comprendre les détails, mais ce fichier contient au moins :

— La commande pour démarrer le démon (`ExecStart`)
— La commande pour arrêter le démon (`ExecStop`)
— Un descriptif (`Description`)
— Les dépendances (`After`, `WantedBy`)

Q9. Pour lister les services disponibles, lancez la commande `systemctl`. Retrouvez la ligne concernant nginx avec `systemctl | grep nginx`.

Q10. Comment savoir si le serveur ssh est lancé (c'est-à-dire, comment connaître le status du serveur ssh) ?

1.9 Les logs !

Q1. Naviguez dans `/var/log`. Quels sont tous ces fichiers, leurs droits et que contiennent-ils ?

Q2. Lancez la commande `tail -f /var/log/nginx/access.log`, puis, pendant que cette commande tourne, rechargez la page d'accueil de nginx. Vous devriez voir apparaître une ligne par requête dans le fichier de log (`tail -f` les affiche au fur-et-à-mesure).

Q3. Chargez la page `http://IP_VM/404/`. Comme cette page n'existe pas, vous verrez apparaître la page d'erreur 404 (Not Found) de nginx.

Q4. Installez maintenant le paquet `apache2`. Ce paquet correspond à la version 2 du serveur web Apache httpd. C'est un concurrent direct de nginx, qui va essayez d'écouter sur le port 80 (Ce qui est impossible pour l'instant car nginx est déjà en écoute dessus, mais faisons semblant de ne pas savoir pourquoi un instant ...).

Q5. Forcez un lancement d'apache :

```
sudo systemctl start apache2
```

Q6. Essayez de recharger la page d'erreur 404 : c'est toujours nginx qui répond !

Q7. Regardez le status du service apache2 avec `systemctl status`. Vous devriez obtenir « Active : inactive (dead) », et la commande vous donne quelques lignes de logs qui devraient vous mettre sur la voie de la raison de l'échec du lancement. Vous pouvez retrouver les détails avec la commande `journalctl`.

Q8. Devant l'échec de cette tentative de lancer deux serveurs web sur la même machine, nous décidons d'abandonner et de désinstaller apache.

2 Si le temps le permet : copie de fichiers à distance (vu en LIFSE en L2)

Vous devez apprendre à utiliser ssh pour copier des fichiers à distance. Une alternative est `rsync`.

Q1. Faites le en utilisant la ligne de commande, via `scp`, `pscp` (`PuTTY`) ou `rsync`.
Certains logiciels reposant sur la bibliothèque `FUSE` (File System in User Space) permettent de configurer un système de fichiers en réseau en tant que simple utilisateur. Vous pourrez en installer et utiliser un pour associer à un répertoire du compte etudiant de la VM un répertoire que vous utilisez à l'université pour les TPs de LIFPCA.

Q2. Qu'est-ce qu'un disque réseau ? Donner un exemple de logiciel utilisant ce type de bibliothèque.

Q3. Installez le logiciel `sshfs`.

Q4. Regardez le manuel de la commande `sshfs`, utilisez-la pour que le répertoire `votrevm:/home/etudiant/univ/` corresponde au répertoire `"$HOME"/LIFPCA/VM` sur votre poste de travail de l'université.

Q5. Quel est l'intérêt de faire des systèmes de fichiers dans « l'espace utilisateur » ?

3 Pour aller plus loin

3.1 Configuration de sudo

`sudo` permet d'autoriser à un utilisateur l'exécution d'une ou plusieurs commandes avec des droits privilégiés. C'est une commande **additionnelle** au système.

Q1. Comment la machine ASR7, dont vous avez obtenu une copie pour créer votre VM, a été configurée pour donner la possibilité à `etudiant` de lancer n'importe quelle commande avec `sudo` ?

Q2. Comment configurer `sudo` pour donner le droit à `lurong` d'exécuter la commande `cat` avec les droits `root` (afin par exemple de pouvoir afficher le contenu de `/etc/shadow`) ?

Q3. Comment se fait-il que la commande `sudo` puisse donner le droit à un utilisateur de lancer un processus (donc à priori avec les droits propres à cet utilisateur) avec des droits `root` ?

3.2 Les "fichiers" du noyau Linux

Q1. Naviguez dans `/boot`, `/usr/src/`, `/lib/modules/`
Peut-on lire la même chose en local ? Quelles sont les capacités du noyau chargé ?

Q2. Faire `dmesg`, distant et local.
Repérer le type de processeur, et de disque dur.
Faites `cat /proc/cpuinfo` et `cat /proc/meminfo`
Quelles informations obtenez-vous ?
Peut-on les obtenir en local et/ou distant ?
Du coup, quel utilisateur peut exécuter la commande ?
Comment était-il possible de le savoir avant ?

Q3. Listez `/proc/` et commentez ce que vous y trouver.
Des remarques sur `/proc/sys/net/ipv4/ip_forward` ?

Q4. Quels systèmes de fichiers le noyau installé est-il capable de lire ?

Q5. Qu'est-ce que `/dev/` ?
Qu'est-ce que `/dev/shm/` ? Qu'est-ce que `/dev/disk/by-uuid/` ?

3.3 Jouons à casser notre environnement

Comme le titre l'indique, ne pas faire ces manipulations sur votre compte habituel pour ne pas prendre de risque.

— Exécutez la commande `PATH=` (équivalente à `PATH=''`, c'est-à-dire avec une chaîne vide à droite du =), puis essayez `ls` et `cd`. Expliquez ce qu'il se passe.

3.4 Préserver une session de la déconnexion avec `screen`

Un scénario assez classique :

- Un utilisateur lance une commande sur son serveur
- L'heure tourne, l'utilisateur doit rentrer chez lui
- De chez lui, l'utilisateur se reconnecte à son serveur via SSH, et aimerait reprendre la main sur ce qu'il a démarré avant de partir.

La commande `screen` (ou un de ses petits frères comme `tmux` ou le couple `dvtm/dtach`) permet de répondre à ce problème (et bien d'autres comme la possibilité de découper votre terminal en sous-fenêtres).

- Dans un terminal lancé sur votre VM, entrez la commande `screen`, et validez le message d'accueil avec Entrée si besoin.
- Entrez quelques commandes : tout se passe normalement.
- Entrez Control-a puis d. Vous devriez revenir au shell initial, mais ce que vous avez démarré dans screen tourne toujours. Si un calcul est en cours, le calcul continue.
- Entrez la commande `screen -r` : votre environnement screen est de retour.
- Refaites Control-a puis d. Déconnectez-vous complètement de la VM.
- (Pour respecter strictement le sénario ci-dessus, il faudrait rentrer chez vous ici ...)
- Reconnectez-vous via SSH et faites `screen -r` : votre environnement est encore là.

Une alternative est la commande `nohup` qui permet de lancer une commande qui survit à la fin de la session de l'utilisateur courant (mais ne permet pas de reprendre la main facilement dessus).

Annexe 1 – Accès à distance

Voici quelques informations pour vous connecter à distance sur les machines de l'université et sur la VM qui servira pour le TP d'administration système.

Pour l'exemple, nous allons essayer de nous connecter au PC `b710l0201.univ-lyon1.fr` qui se trouve physiquement dans une salle machine (sous réserve que ce PC soit allumé). S'il ne fonctionne pas, essayez de deviner l'adresse d'un autre PC du nautibus, les noms sont toujours du type `b710` (fut un temps ou le Nautibus s'appelait « bâtiment 710 »...), la lettre « l » (pas le chiffre 1), puis 2 chiffres pour la salle (par exemple 02 pour la salle Nautibus TP2), et deux chiffres pour le numéro de poste (par exemple 01 pour la première machine de la salle).

Dans les explications qui suivent, on suppose que vous utilisez une machine de type Unix (Linux, le sous-système Linux de Windows, ou Mac OS). Si vous êtes sous Windows, des explications pour utiliser MobaXTerm pour faire les manipulations équivalentes sont disponibles ici : `https://nlouvet.gitlabpages.inria.fr/lifasr5/connec.html`.

Note

Dans un monde sans firewall, on pourrait s'y connecter simplement avec la commande :

```
ssh votre-login-lyon1@b710l0201.univ-lyon1.fr
```

En pratique, ça ne marchera pas de l'extérieur, car l'accès est bloqué pour des raisons de sécurité. En Master, les étudiants ont accès au VPN qui permet de « faire comme si » la machine faisait partie du réseau local, mais ce n'est pas le cas en Licence. C'est pour cette raison que nous allons voir quelques solutions alternatives.

A Rebond SSH à la main sur `linuxetu`

Même depuis l'extérieur, vous avez accès via SSH à la machine `linuxetu.univ-lyon1.fr`, donc vous pouvez vous connecter d'abord à `linuxetu`, puis dans le shell ouvert sur `linuxetu` lancer une connexion SSH vers votre machine virtuelle. Par exemple, si votre login Lyon 1 est `p1234567`, vous pouvez le faire comme ceci :

```
votre-ordinateur$ ssh p1234567@linuxetu.univ-lyon1.fr
p1234567@linuxetu.univ-lyon1.fr's password: <votre-mot-de-passe-Lyon1>
[...]
==============
Ceci est une passerelle SSH.
Votre 'home universitaire' est dans le dossier HOMELYON1.
Si vous utilisez une clé SSH, vous pouvez exécuter la commande mhome pour le monter.
==============

p1234567@linuxetu:~$ ssh p1234567@b710l0202.univ-lyon1.fr
The authenticity of host b710l0202.univ-lyon1.fr can't be established.
ECDSA key fingerprint is SHA256:/VAtIv4LXOEiKzQRNYsRPcxENUvTrsuAvK31gKp8e+s.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.75.165' (ECDSA) to the list of known hosts.
chaprot@192.168.75.165's password: <mot-de-passe-de-la-vm>
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.4.0-22-generic x86_64)
[...]
chaprot@vm-26:~$
```

L'ordinateur vous demande de vérifier l'empreinte de la clé, nous supposerons que tout est bon.

B Rebond SSH automatique sur `linuxetu`

Sur la machine locale, on peut demander à faire passer automatiquement les connections par la machine `linuxetu`. Pour cela, dans le fichier `~/.ssh/config` (*à créer s'il n'existe pas*), ajoutez les lignes (remplacez votre-login-lyon1 bien sûr) :

```
Host !forge.univ-lyon1.fr !linuxetu.univ-lyon1.fr !*.tpr.univ-lyon1.fr *.univ-lyon1.fr
ProxyCommand ssh -N -W %h:%p votre-login-lyon1@linuxetu.univ-lyon1.fr
```

On peut lire ces lignes comme « Pour toutes les machines, à l'exception de la forge, de linuxetu, et des machines des salles TPR, dont le nom termine par univ-lyon1.fr, ouvrir une connexion avec `linuxetu` pour créer la connexion demandée. ». Si vous ne vous êtes pas trompés, vous devriez pouvoir ouvrir une connexion SSH avec simplement :

```
ssh votre-login-lyon1@b710l0202.univ-lyon1.fr
```

Pour pouvoir vous connecter à votre VM à partir de son adresse IP, il faudra ajouter l'adresse IP de la VM à la liste dans la ligne `Host`. La commande devrait vous demander deux fois votre mot de passe : une fois pour `linuxetu` et une autre pour `b710l0202.univ-lyon1.fr`.

Pour appliquer la même chose en vous connectant à une machine via son adresse IP, on peut faire :

```
Host 192.168.x.y  # Remplacer 192.168.x.y par la vraie IP
ProxyCommand ssh -N -W %h:%p votre-login-lyon1@linuxetu.univ-lyon1.fr
```

C Tunnel SSH

Mise en place du tunnel ssh (à lancer depuis votre machine de travail, remplacer `IP_VM` par l'adresse IP de votre machine virtuelle et `login` par votre login Lyon 1) :

```
ssh -L 20002:b710l0202.univ-lyon1.fr:22 votre-login-lyon1@linuxetu.univ-lyon1.fr
```

Dans un autre terminal, pour accéder à la machine :

```
ssh -p 20002 login@localhost
```

Modifier le numéro du port (ici 20002) au besoin !

D Transfert de fichiers

Les explications qui précèdent vous permettent d'exécuter des commandes à distance sur une machine. Pour transférer des fichiers depuis votre compte Lyon 1 vers votre machine personnelle (et l'inverse), une documentation est disponible ici (FileZilla, rsync, accès web, ...) :

`https://forge.univ-lyon1.fr/dpt-info/docs/-/blob/master/fichiers-distants.md`

Annexe 2 – Accès aux machines virtuelles (VMs)

On peut noter que la VM n'a qu'un CPU (VCPU, pour Virtual CPU), même si elle tourne en réalité sur un serveur physique qui en a beaucoup plus que ça. Tout se passera donc comme si nous étions sur une machine mono-cœur, ce qui simplifie un peu les choses en terme d'ordonnancement.

A Se connecter à sa VM

Q1. Connectez-vous en utilisant l'utilisateur `etudiant` (mot de passe : `etudiant`). Depuis l'intérieur de l'université, on peut utiliser SSH normalement comme ceci :

```
ssh etudiant@IP_VM
```

Voici la procédure

1. saisir le mot de passe "etudiant" ⇝ le prompt indique un changement de mot de passe obligatoire et demande de saisir le mot de passe actuel
2. saisir le mot de passe "etudiant" (une seconde fois donc)
3. saisir le nouveau mot de passe
4. ressaisir le nouveau mot de passe ⇝ le serveur termine la connexion
5. `ssh etudiant@IP_VM`
6. saisir le nouveau mot de passe

Note

Vous pouvez aussi utiliser le login `chaprot` avec le mot de passe `lif12`.

Note

Si vous travaillez de l'extérieur de l'université, alors malheureusement la machine virtuelle n'est pas accessible directement. Si vous avez installé le VPN Lyon 1, lancez-le et connectez-vous comme si vous étiez à l'extérieur de l'université. Sinon, utilisez un rebond SSH sur `linuxetu` avec l'une des méthodes possibles (en utilisant l'IP de votre VM comme nom de machine et le login `etudiant`).
Par exemple, la connexion avec rebond SSH manuel ressemble à ceci :

```
moy@moylip:~$$ ssh p1234567@linuxetu.univ-lyon1.fr
p1234567@linuxetu.univ-lyon1.fr's password: <mot-de-passe-Lyon1>
[...]
matthieu.moy@linuxetu:~$ ssh etudiant@IP_VM
etudiant@IP_VM's password: <mot-de-passe-de-etudiant>
[...]
etudiant@vm-26:~$
```

Q2. La commande SSH vous a donné un accès shell, en mode texte. Les outils graphiques dont vous avez probablement l'habitude ne sont donc pas disponibles. La commande SSH permet en général d'utiliser des commandes graphiques si on utilise `ssh -X` pour se connecter au serveur, mais cela ne marche pas sur `linuxetu`. Profitons-en pour nous familiariser avec des éditeurs de textes en mode texte : `nano` est un éditeur peu avancé mais simple à utiliser. `emacs` et `vim` sont deux éditeurs très avancées mais demandant une phase d'apprentissage relativement longue. Dans les trois cas, vous pouvez ouvrir un fichier simplement en appelant l'éditeur avec le nom du fichier en paramètre sur la ligne de commande, par exemple `nano lancement.cpp`.

Q3. Récupérez maintenant le fichier `lancement.cpp` sur votre VM. Une manière de faire est d'utiliser la commande `wget` (qui télécharge un fichier) depuis la VM :

```
wget url_vers_lancement.cpp
```

Une autre est de télécharger le fichier sur votre PC physique, puis de l'envoyer à la VM avec une commande comme :

```
rsync -av lancement.cpp etudiant@IP_VM:
```

Malheureusement, cette commande ne fonctionnera pas directement depuis l'extérieur de l'université sans VPN ou ajout dans le `~/.ssh/config` comme indiqué ci-dessus.

Q4. Vous pouvez passer root via la commande `sudo su`. Il est conseillé de travailler comme utilisateur normal (`etudiant`), et de ne passer root que quand c'est nécessaire, c'est à dire pour lancer l'exécutable de votre TP pour le cas qui nous intéresse. Par exemple :

```
$ nano lancement.cpp
$ g++ -std=c++11 -pthread lancement.cpp  -o lancement
$ sudo ./lancement
```

Attention : la VM que vous avez créée pourra être supprimée sans avertissement. Si vous réalisez un TP, à la fin de celui-ci, récupérez vos données sur votre compte (par exemple avec la commande `rsync`).

B Un peu de confort avec VSCode

Si vous avez vos habitudes avec un éditeur graphique avancé (comme VSCode), vous aurez peut-être envie de les conserver en travaillant à distance. Installer VSCode sur la VM et l'utiliser à distance serait pénible : beaucoup d'espace disque gaspillé sur la VM, et l'affichage graphique déporté est assez lent. Nous allons faire autrement : utiliser VSCode en local, mais lui demander d'accéder aux fichiers à distance via SSH.

- Installer l'extension *Remote - SSH* [1]
- Configurez le rebond SSH automatique dans `~/.ssh/config` si ce n'est pas déjà fait (cf. section B)
- Cliquer sur l'icône « Open a remote window » avec une icône ressemblant à >< en bas à gauche de l'écran, ou faites Control+Shift+P puis entrez la commande `>Remote-SSH: Connect to Host...`
- Entrer `etudiant@IP_VM` et validez. Entrer vos mots de passes quand VSCode vous les demande.
- Et voilà, vous avez une fenêtre VSCode connectée à votre VM. Vous pouvez l'utiliser presque comme si vous étiez en local sur la VM.

Plus d'informations : `https://code.visualstudio.com/docs/remote/ssh`.

1. `https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh`

C Créer sa propre VM

Pour créer une VM, vous devez vous connecter à l'interface d'administration :
`http://cloud-info.univ-lyon1.fr/` (inaccessible depuis l'extérieur du réseau de l'université) et utiliser le domaine `UNIV-LYON1.fr`, avec vos identifiants habituels Lyon 1.

Vous faites partie du projet `OpenStack` ASR7 (si besoin, sélectionnez-le depuis le menu déroulant en haut de l'écran). Vous devez bien faire attention à ne pas créer trop de machines, ni utiliser trop de ressources.

Vous pouvez créer une instance de votre machine grâce à :
`Projet->Calcul->Instances->Lancer une instance`.

Faites attention à :

- donner un nom reconnaissable pour votre instance ;
- utiliser l'image `Ubuntu Server 22.04.1 LTS - Login etudiant` ;
- utiliser le gabarit (ensemble de ressources) `m1.tiny` ;
- ne créer qu'une seule instance ;
- lancer la création.

La machine virtuelle va être créée (cela demande un peu de temps) et apparaître dans la liste des Instances. Elle obtiendra une **adresse IP** que vous noterez.