SUD341 : systèmes temps réel et embarqués

TP3–le temps réel sous LINUX: L'API POSIX

Prof. Abdeslam EN-NOUAARY

Introduction

L'objectif de ce TP est d'apprendre le développement en temps réel sous les systèmes d'exploitation de type UNIX/LINUX à travers l'interface de programmation standard POSIX (*Portable Operating System Interface for UNIX*) définie depuis 1988 par IEEE sur proposition de Monsieur Richard Mathew Stallman. La norme présente principalement une bibliothèque riche *pthreads* qui facilite la programmation concurrentielle/multitâches ainsi que les aspects s'y relatant tels que la communication et la synchronisation. Des exemples seront utilisés pour illustrer les différents mécanismes présentés, ainsi que des extensions à faire sont aussi demandées pour approfondir davantage les connaissances acquises.

I- La norme POSIX

POSIX définit un ensemble de normes pour assurer le développement d'applications portables au niveau du code source C, entre les systèmes d'exploitation conformes à la norme POSIX. Autrement dit, POSIX est le standard officiel qui définit les interfaces communes à tous les systèmes de type UNIX :Quand on veut qu'un programme C fonctionne sur tous les UNIX alors on doit le coder en respectant les interfaces POSIX et voilà magiquement il sera compatible avec LINUX, le BSD, le System V, etc. Bien entendu c'est une contrainte puisqu'on doit se limiter au plus petit dénominateur commun et on ne peut plus utiliser les spécificités techniques de chaque plateforme, à moins de devoir faire des chemins spécifiques dans le code pour chacun des OS mais au détriment entre autres du temps de développement, de facilité de relecture, et du nombre de bugs.

POSIX a été standardisée par IEEE, ANSI et ISO sous le nom POSIX 1003.1X (POSIX.1003.1a : norme de base, POSIX.1003.1b : extension temps réel, POSIX.1003.1c : extension threads, etc.). La norme POSIX définit notamment :

- Les commandes Shell de base (ksh, ls, man, ..).
- L'API des appels système.
- L'API des threads.

Dans ce TP, nous allons nous intéresser surtout à la bibliothèque *pthreads* de la norme POSIX qui nous fournit un ensemble de fonctions intéressantes pour la programmation temps réel surtout la concurrence/multitâches, et la communication et la synchronisation entre les tâches.

II. Les threads POSIX (ou les pthreads)

Les mécanismes de création de processus de style UNIX (fork, wait, pipe, etc.) sont très coûteux et ne peuvent être utilisés que pour des programmes nécessitant un nombre restreint de processus lourds. Pour pallier ce problème, diverses bibliothèques de threads poids légers furent introduites depuis le milieu des années 90 dont la plus connue est la bibliothèque de `threads POSIX` (ou simplement les `pthreads`). La bibliothèque `pthreads` définit des douzaines de fonctions, dans le cadre du langage C, pour la programmation concurrente avec les threads. Un thread (un processus léger ou un fil d'exécution en français) est une partie du code d'un programme (une fonction), qui se déroule parallèlement à d'autres parties du programme. Au sein d'un processus, des threads peuvent exécuter, de façon indépendante, des séquences d'instructions en parallèle.

Contrairement à un processus lourd qui, lorsqu'il est créé, possède sa propre zone de ressources contenant les descripteurs de fichiers, les tables des pages mémoire et une pile interne qui lui sont toutes dédiées, les threads, lorsqu'ils sont créés, partagent le même espace mémoire et la même table des fichiers ouverts mais chacun dispose de sa propre pile (voir figure 1). L'avantage des processus légers sur les processus lourd est qu'ils sont très favorables à la programmation multitâches, car la création d'une tâche est moins coûteuse en termes de ressources du fait qu'elles ne sont pas toutes dupliquées. De plus, l'utilisation des tâches simplifie la gestion des problématiques liées à l'exécution concurrente telles que la communication et la synchronisation entre les tâches.

Cependant, puis que les accès aux données étant directs et des accès concurrents étant possibles, les données peuvent être incorrectes si des précautions ne sont pas prises. Il existe donc des mécanismes de protection qui permettent de réguler l'accès aux données, tels que les sémaphores.

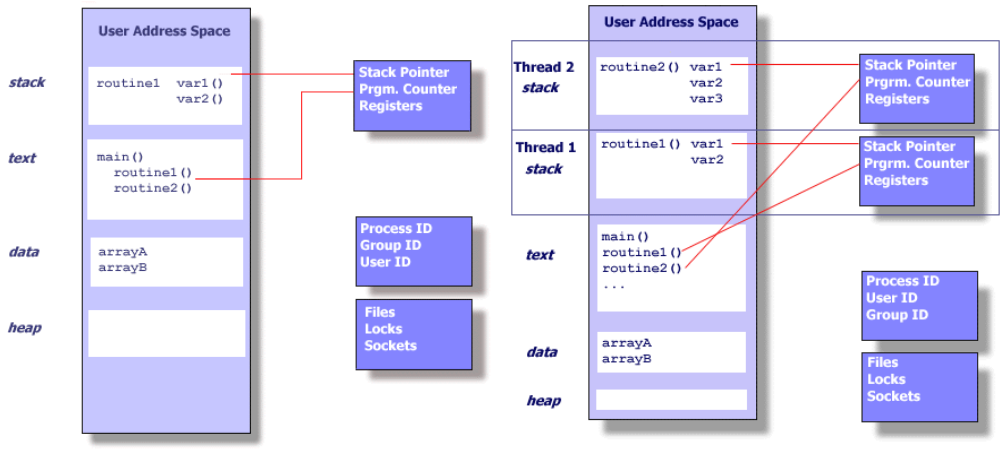

Figure 1. Processus versus thread de point de vue image mémoire

Dans ce qui suit, nous passons en revue les fonctions offertes par la librairie `pthreads` pour la gestion des threads POSIX. Ces fonctions sont définies dans la bibliothèque `pthread.h`. Ainsi, pour utiliser les threads dans un programme :

- Il faut d'abord inclure la librairie `pthread`

```
#include <pthread.h>
```

- Il faut ensuite compiler avec l'argument `-lpthread`

```
gcc exple.c -o exple -lpthread
```

a) Création d'un thread

Pour créer un thread, on doit utiliser la fonction suivante:

```
int pthread_create(pthread_t *thread_id, pthread_attr_t *thread_attr,
                   void *(*thread_func)(void *), void *arg);
```

Le premier argument `thread_id` est l'adresse du descripteur du thread créé, le deuxième argument correspond à des attributs que l'on peut donner au thread créé (comportement en fin d'exécution, politique d'ordonnancement, possibilité d'annulation, etc.); on passera généralement `NULL` pour ce paramètre `thread_attr` afin de spécifier que les attributs par défaut peuvent être utilisés, le troisième argument est le nom de la fonction à exécuter au lancement du thread; il doit être de type `void *thread_func(void *arg)`, et le quatrième argument correspond à l'argument passé à la fonction `thread_func` lors de l'appel; il doit être de type `void *` (cet argument peut aussi être `NULL`). Si la création est effectuée correctement, la fonction renvoie 0, sinon un code d'erreur. Au moment de l'appel à la fonction `pthread_create`, le thread correspondant est lancé et peut exécuter la fonction `thread_func`.

b) Terminaison d'un thread

Si un thread doit retourner une valeur à la fin de son exécution, il le fera à l'aide de la fonction `pthread_exit` utilisée de la manière suivante:

```
void pthread_exit(void *value_returned);
```

Le paramètre `value_returned` correspond à la valeur qui doit être retournée après avoir été transtypée en `(void *)`.

c) Attente d'un thread

Le processus qui exécute le `main` est aussi un thread et s'appelle le thread principal. Le thread principal ou tout autre thread peut attendre la fin de l'exécution d'un autre thread par la fonction `pthread_join`. Cette fonction permet aussi de récupérer la valeur retournée par la fonction du thread à attendre; c'est-à-dire, la valeur de la fonction `pthread_exit`. Le prototype de la fonction `pthread_join` est le suivant :

```
int pthread_join(pthread_t thread_id,void **value_returned);
```

Le premier paramètre est l'identifiant du thread (que l'on obtient par `pthread_create`), et le second paramètre est un passage par adresse d'un pointeur qui permet de récupérer la valeur retournée par `pthread_exit`.

La fonction `pthread_join` a principalement les deux effets suivants:

- bloquer le processus appelant jusqu'à la terminaison du thread d'identifiant `thread_id`;
- récupérer la valeur retournée par le thread et la stocker dans `value_returned`.

d) Annulation d'un thread

Il est possible de terminer un thread par un appel particulier à partir d'un autre thread:

```
int pthread_cancel(pthread_t thread_id);
```

Le thread qui va être annulé invoque alors automatiquement la fonction `pthread_exit` en renvoyant une valeur spéciale `PTHREAD_CANCELED`.

Exemple

Le programme que je vous donnerai est une illustration de la gestion des threads POSIX. Il permet de créer et lancer deux threads concurrents. Le premier thread permet d'afficher un message de bienvenue et de lire un entier au clavier puis de le retourner au thread principal pour affichage sur la sortie standard.

1) Exécuter le programme et analyser son code et le résultat obtenu.

2) Ajouter d'autres threads dans le programme pour afficher l'entier lu aussi bien en décimal qu'en hexadécimal et qu'en octal. Penser à utiliser le design pattern Strategy

3) Modifier le programme pour qu'il affiche le temps pris par chaque opération (lecture, affichage, etc.) ainsi que son temps de réponse depuis son lancement jusqu'à terminaison.

III. Synchronisation des pthreads

Rappelons que lorsqu'un nouveau processus est créé par un fork, toutes les données (variables globales, variables locales, mémoire allouée dynamiquement), sont dupliquées et copiées, et le processus père et le processus fils travaillent ensuite sur des variables différentes. Cependant, dans le cas de threads, la mémoire est partagée, c'est à dire que les variables globales sont partagées entre les différents threads qui s'exécutent en parallèle. Cela pose des problèmes lorsque par exemple deux threads différents essaient d'écrire et de lire une même donnée. Deux types de problèmes peuvent alors se poser :

- Deux threads concurrents essaient en même temps de modifier une variable globale;
- Un thread modifie une structure de donnée tandis qu'un autre thread essaie de la lire.

Ces deux situations compromettent l'état du système et donnent lieu à des données incohérentes qui ne peuvent plus être utilisées surtout dans des systèmes critiques temps réel. On parle aussi dans ce cas d'une situation de compétition (″race condition″ en anglais); il s'agit d'une situation où des actions effectuées par des unités d'exécution différentes s'enchaînent dans un ordre illogique, entraînant des états non prévus.

Pour remédier à ce problème, il faut donc avoir recours à un mécanisme de synchronisation qui fait que les threads ne peuvent pas par exemple accéder en même temps à une donnée ou à une ressource partagée; c'est-à-dire, procéder par l'exclusion mutuelle.

En programmation concurrente/multithreads, la synchronisation se réfère à deux concepts distincts mais liés : la synchronisation de *threads* et la synchronisation de *données*. La synchronisation de threads est un mécanisme qui vise à bloquer l'exécution de certains threads à des points précis de leur flux d'exécution, de manière que tous les threads se rejoignent à des étapes relais données, tel que prévu par le programmeur. La synchronisation de données, quant à elle, est un mécanisme qui vise à conserver la cohérence des données dans un environnement multitâches. Initialement, la notion de synchronisation est apparue pour la synchronisation de données.

Il existe principalement trois mécanismes pour assurer la synchronisation entre les `pthreads`, à savoir : les mutexes, les sémaphores, et les variables de conditions.

a) Les mutexes

Les threads POSIX fonctionnent de manière asynchrone et peuvent donc être synchronisés à l'aide de mutexes (ou verrous en français). Les threads partageant entre eux les variables globales et les descripteurs de fichiers ouverts, une gestion de l'accès concurrentiel par mutex doit donc être introduite pour garantir la cohérence de données qui résulte d'une section critique mal contrôlée. Les mutexes sont des verrous proposés par la bibliothèque `Pthreads;` pour les utiliser, il faut tout d'abord inclure ladite bibliothèque :

```
#include <pthread.h>
```

La déclaration d'un mutex se fait de la façon suivante :

```
pthread_mutex_t mutex;
```

L'initialisation du mutex se fait avant la création des threads de la manière suivante:

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);
```

Le premier paramètre sera rempli à l'initialisation et sera utilisé comme identifiant pour ce mutex. Le deuxième paramètre permet de choisir un certain nombre d'attributs pour le mutex initialisé (par exemple le type de mutex rapide ou recursif). Si les attributs par défaut doivent être utilisés, on placera cette valeur à NULL. Lorsqu'un thread souhaite accéder à une section critique, il verrouille le mutex correspondant de la manière suivante:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Si un autre thread se trouve dans la section critique, le thread appelant `pthread_mutex_lock` sera bloqué jusqu'à la libération du mutex par la fonction suivante :

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Il est également possible pour un thread d'utiliser la fonction `pthread_mutex_trylock` afin de tester l'accès à une section critique sans être bloqué si celle-ci est déjà utilisée par un autre thread.

b) Les sémaphores

Rappelons qu'une section critique est, en général, une partie du code où un processus ou un thread ne peut rentrer qu'à une certaine condition. Lorsqu'un processus/thread entre dans la section critique, il modifie la condition pour les autres processus/threads. Par exemple, si une section du code ne doit pas être exécutée simultanément par plus de `n` threads alors chaque fois qu'un thread veut rentrer dans la section critique il doit vérifier qu'au plus `(n-1)` threads y sont déjà. Lorsqu'un thread entre dans la section critique, il modifie la condition sur le nombre de threads qui se trouvent dans la section critique. Ainsi, un autre thread peut se trouver empêché d'entrer dans la section critique si le nombre d'autorisations est déjà atteint.

La difficulté d'implémenter correctement une section critique est qu'on ne peut pas utiliser une simple variable comme un compteur. En effet, si le test sur le nombre de threads et la modification du nombre de threads lors de l'entrée dans la section critique se font séquentiellement par deux instructions, il ne serait pas surprenant qu'un autre thread pourrait tester la condition sur le nombre de threads justement entre l'exécution de ces deux instructions, et deux threads passeraient en même temps dans la section critique. Il y a donc nécessité de tester et modifier la condition de manière atomique, c'est-à-dire qu'aucun autre processus/thread ne peut rien exécuter entre le test et la modification. C'est une opération atomique appelée `Test and Set Lock`.

Pour contrôler les accès à une section critique (contenant un objet partagé que ce soit logique ou physique), Edgar Dijkstra suggéra en 1965 l'emploi d'un nouveau type de variables appelées les sémaphores. Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès à une section critique. Il a donc un nom et une valeur initiale, par exemple semaphore `S = 10`.

Les sémaphores sont manipulés au moyen des opérations `P` ou `wait` et `V` ou `signal`. L'opération `P(S)` décrémente la valeur du sémaphore `S` si cette dernière est supérieure à `0`; sinon le processus appelant est mis en attente. L'opération `V(S)` incrémente la valeur du sémaphore `S` si aucun processus n'est bloqué par l'opération `P(S)`; sinon, l'un d'entre eux sera choisi et redeviendra prêt. Le test du sémaphore, le changement de sa valeur, et le réveil et la mise en attente éventuelle d'un thread sont effectués en une seule opération atomique indivisible.

Les sémaphores sont un type `sem_t` et un ensemble de primitives de base qui permettent d'implémenter des conditions assez générales sur les sections critiques. Un sémaphore possède un compteur dont la valeur est un entier positif ou nul. On entre dans une section critique si la valeur du compteur est strictement positive. Pour utiliser un sémaphore, on doit tout d'abord inclure la bibliothèque spécifique :

```
#include <semaphore.h>
```

Ensuite, il faut créer et initialiser le sémaphore avec la fonction :

```
int sem_init (sem_t *sem, int shared, unsigned int value);
```

Le premier argument est un passage par adresse du sémaphore, le deuxième argument indique si le sémaphore peut être partagé par plusieurs processus, ou seulement par les threads du processus appelant (la valeur `shared` égale `0`). Enfin, le troisième argument est la valeur initiale du sémaphore. Après utilisation, il faut systématiquement libérer le sémaphore avec la fonction :

```
int sem_destroy (sem_t *sem);
```

Le reste des primitives de base sur les sémaphores se présentent comme suit :

- `sem_wait`: le thread appelant teste et se bloque si le sémaphore est nul; sinon il décrémente le compteur et passe à l'instruction suivante faisant généralement partie de la section critique :

  ```
  int sem_wait (sem_t *sem);
  ```

- `sem_post`: le thread appelant incrémente le compteur si aucun thread n'est bloqué en attente d'autorisation d'accéder à la section critique; sinon, l'un de threads bloqués sera réveillé pour procéder à la section critique :

  ```
  int sem_post (sem_t *sem);
  ```

- `sem_getvalue`: le thread appelant récupère la valeur du compteur dans une variable passée par adresse :

  ```
  int sem_getvalue (sem_t *sem, int *value);
  ```

- `sem_trywait`: le thread appelant teste si le sémaphore est non nul et décrémente le sémaphore mais sans se bloquer si ce dernier est nul. Dans ce dernier cas, la fonction retourne une erreur. Il faut utiliser cette fonction avec précaution car elle risque d'être la source de plusieurs bugs difficiles à corriger :

```
int sem_trywait(sem_t *sem);
```

c) Les variables de conditions

Outre les mutexes et les sémaphores discutés précédemment, POSIX met un autre moyen à la disposition du développeur pour synchroniser des threads: les variables de condition de type `pthread_cond_t`. De telles variables permettent de mettre en attente un ou plusieurs threads jusqu'à ce qu'un événement donné se produise. L'initialisation de la condition se fait de la manière suivante:

```
int pthread_cond_init(pthread_cond_t *condition, pthread_condattr_t *attr);
```

Le premier paramètre sera rempli à l'initialisation et sera utilisé comme identifiant pour la condition. Le deuxième paramètre permet de choisir un certain nombre d'attributs pour la condition initialisée. Si les attributs par défaut doivent être utilisés, on placera cette valeur à `NULL`. Lorsqu'un thread souhaite attendre une condition, il effectue cette série d'appel :

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_cond_wait(pthread_cond_t *condition, pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

L'appel à `pthread_cond_wait` doit être protégé par un mutex. Le mutex doit être passé comme deuxième paramètre à la fonction `pthread_cond_wait`. Ceci permet de déverrouiller le mutex et laisser le thread notifiant l'événement accéder à la variable de condition. Lorsqu'un thread constate la réalisation de l'événement, une notification est alors envoyée à un ou à tous les threads en attente de l'événement.

Ainsi, si on veut réveiller un seul thread bloqué sur une variable de condition, on doit faire :

```
int pthread_cond_signal(pthread_cond_t *condition);
```

Là aussi il est recommande d'appeler cette fonction au sein d'une section critique protégée par le mutex utilisé par le thread en attente.

Si par contre, on veut réveiller tous les threads en attente, on doit plutôt utiliser la fonction :

```
int pthread_cond_broadcast(pthread_cond_t *condition);
```

A noter que les variables de conditions ne peuvent pas être utilisées toutes seules; il faut toujours

les combiner avec des conditions sur l'état du programme qui en gros assurent que l'appel à `pthread_cond_wait` se fait toujours avant l'appel à `pthread_cond_signal`.

Une variable de condition qui n'est plus utilisée doit être libérée par:

```
int pthread_cond_destroy(pthread_cond_t *condition);
```

Exemple

Le programme que je vous donnerai ici permet d'illustrer la synchronisation entre les threads POSIX en utilisant des mutexes et des variables de conditions. Le but du programme est de gérer des transactions bancaires sur un compte qui est alimenté régulièrement pour accommoder les dépenses personnelles quotidiennes d'un couple qui fixent le plafond de ces dépenses à 1000 dirhams. Bien entendu, le solde du compte ne doit jamais descendre au dessous de zéro (la banque ne veut plus de comptes à découvert !!!). A chaque alimentation du compte, on le crédite d'un montant de 200 dirhams. Quant aux dépenses, le montant de chacune d'elles est une valeur générée aléatoirement entre 1 et 50 dirhams. Nous allons utiliser cinq threads pour simuler les dépenses et un thread pour la banque.

1) Exécuter le programme et analyser le résultat obtenu.
2) Enlever les variables de conditions dans le programme et exécuter-le. Que remarquez-vous ?
3) Enlever les mutexes dans le programme et exécuter-le. Que remarquez-vous ?
4) Observer et analyser le taux d'utilisation du processeur de la machine sur laquelle tourne le programme. Que peut-on conclure ?
5) Peut-on améliorer davantage le programme ? Vérifier si la limite de 1000 dirhams par jour est toujours respectée par l'application.

III. Communication entre les pthreads

Lorsqu'un programme est décomposé en plusieurs threads, ceux-ci ne sont en général pas complètement indépendants les uns des autres mais ils doivent communiquer entre eux pour échanger des données pour que le service demandé par l'utilisateur soit rendu. Cette communication entre threads ainsi que leur synchronisation/coordination constituent un problème complexe dans un environnement multitâches. Dans ce qui suit, nous allons donner un aperçu général sur les principaux mécanismes qu'on peut utiliser pour la communication entre les threads (''InterProcess Communication (IPC)'' en anglais). Nous aurons l'occasion de les présenter tous en détail avec des exemples dans des travaux pratiques futurs.

La première façon pour communiquer avec un thread est d'utiliser les arguments de la fonction de démarrage du thread lors de sa création par la fonction `pthread_create` et aussi la valeur retournée par le thread via l'appel à `pthread_join`. C'est clair qu'il s'agit d'un canal de communication très limité qui ne permet pas l'échange d'information pendant l'exécution du thread. Un exemple de ce mécanisme est déjà présenté auparavant dans ce TP.

La deuxième méthode de faire communiquer les threads est de passer par une mémoire partagée. Il s'agit d'un mécanisme assez facile pour partager de l'information entre les threads d'un même processus ou même appartenant à des processus différents. En effet, tous les threads d'un processus ont accès aux mêmes variables globales et au même heap. Il est donc tout à fait possible pour n'importe quel thread de modifier la valeur d'une variable globale. Deux threads qui réalisent un calcul peuvent donc stocker des résultats intermédiaires dans une variable globale ou un tableau global. Il en va de même pour l'utilisation d'une zone de mémoire allouée par la fonction `malloc`. Ainsi, chaque thread qui dispose d'un pointeur vers cette zone mémoire peut en lire le contenu ou en modifier la valeur.

Le troisième mécanisme de communication entre threads est de passer par les signaux déjà définis par le système d'exploitation. Les signaux sont des primitifs systèmes identifiés par un numéro de 0 à 63, et à chaque signal une application peut y associer une fonction, dite communément handler, qui sera appelée lorsque le signal sera reçu par un thread (l'appel système `signal`). Les signaux sont envoyés par un autre appel système qui prend en paramètres un identifiant du processus cible et le numéro du signal (l'appel système `kill`). Parmi les exemples de signaux on trouve SIGKILL, SIGSTOP, SIGSEGV, ALARM, etc. Il faut noter qu'il est recommandé de ne pas abuser de l'utilisation des signaux car les signaux, en général, ne s'empilent pas : si deux signaux sont envoyés au même processus avant que celui-ci n'ait eu accès au CPU, le comportement n'est pas généralement défini.

Le quatrième mécanisme de communication à citer est simplement d'utiliser les fichiers. Il s'agit du mode de communication le plus ancien entre deux processus/threads et qui peut même fonctionner si les processus sont exécutés consécutivement et non simultanément. Il consiste à écrire les données dans un fichier puis à lire ce fichier. Seul le nom du fichier a besoin d'être connu et il peut être soit fixe par convention au préalable à travers par exemple un fichier de configuration, soit communiqué dans les arguments du programme. Cette communication est simple et souvent utilisée mais possède quand même plusieurs inconvénients :

- l'écriture et la lecture de fichiers sur le disque est une opération coûteuse ce qui peut nuire au temps réel ;
- les données peuvent être interceptées ou modifiées par d'autres processus ou threads ce qui peut compromettre la sécurité de l'application;
- le nom du fichier doit être soit fixe soit communiqué par un autre moyen ce qui limite l'utilisation de fichiers comme moyen de communication entre threads.

Le cinquième moyen de communication entre processus/threads est les pipes (ou tubes en français). Un pipe peut être simplement présenté comme une sorte de tuyau entre deux processus/threads. Un pipe se compose de deux descripteurs de fichiers connectés entre eux : tout ce qui est écrit sur le premier sera lu sur le deuxième. Notons que les pipes sont un moyen de communication à sens unique quoi qu'il est toujours possible de créer deux pipes pour simuler une communication bidirectionnelle entre deux threads.

Le sixième moyen de communication entre processus/threads est les sockets UNIX qui peuvent être considérés similaires aux tubes/pipes nommés. Les principales différences est que les sockets

UNIX sont bidirectionnels, et que les sockets UNIX sont prévues dans une architecture client-serveur, avec un serveur unique qui contrôle le socket, et de nombreux clients qui peuvent communiquer avec lui. En plus des sockets UNIX, il existe le mécanisme de sockets internet qui permettent d'étendre le service de communication à deux threads/processus tournant sur des machines différentes appartenant au même réseau ou à des réseaux différents.

Le septième moyen de communication entre processus/threads est la file de messages (ou MQ : ″Message Queue″ en anglais). Il s'agit d'un mécanisme asynchrone qui utilise des modules temps réel du noyau pour acheminer des informations importantes entre threads, processus, et même entre machines. Comme son nom l'indique, le mécanisme MQ fonctionne comme un service de gestion de files. Les files contiennent des messages avec des priorités variables, et les messages les plus prioritaires et les plus anciens sont lus en premier. Les files sont approvisionnées par un ou plusieurs producteurs/écrivains, et sont lues par un ou plusieurs consommateurs/lecteurs.

Les fonctions pour accéder aux POSIX MQ sont très similaires aux fonctions des autres mécanismes d'IPC tels que les tubes et les sockets, à savoir: ouverture/fermeture, lecture/écriture. La principale différence réside dans le fait que les MQs travaillent avec des messages de longueur fixe ainsi qu'avec un numéro de priorité. Les cinq principales fonctions de base pour la manipulation de POSIX MQ sont : `mq_open`, `mq_receive`, `mq_send`, `mq_close`, et `mq_unlink`.

Exemple

Le programme que je vous donnerai ici illustre l'utilisation des files de messages pour la communication entre les threads/processus. Il s'agit de développer une petite application à deux threads communicants, un émetteur et un récepteur, pour calculer le temps que prend les messages qui transitent entre les deux parties. Pour ce faire, l'émetteur lit l'heure de son système et l'encapsule dans un message puis l'écrit dans la file de messages pour qu'il soit lu par le récepteur. Ce dernier, une fois qu'il lit un message de la file, lit aussi l'heure de son système puis calcule la différence entre les deux quantités pour estimer le temps minimum, le temps maximum, et le temps moyen que prend un message lors de son transit sur le canal de communication.

1) Exécuter le programme et analyser le résultat obtenu.
2) Que pensez-vous de l'efficacité des files de messages, en termes du délai de transmission, par rapport aux autres mécanismes d'IPC tels que les signaux, les mémoires partagées, les pipes, et les sockets UNIX?

III. Étude de cas

Cette section permet de mettre en pratique la quasi-totalité des concepts décrits précédemment et ce en développant une application permettant de traiter des données massives (c'est-à-dire, du BigData) à travers une grande matrice carrée (`n x n`) dont les valeurs représentent des températures collectées en temps réel en interrogeant régulièrement, pendant une période donnée, des capteurs déployés dans une zone géographique sensible. L'objectif est de calculer en temps réel des statistiques concernant les températures collectées telles que la température moyenne, la température médiane, l'écart-type et la variance de températures, et de les afficher sous forme

d'un tableau de bord pour une analyse et une prise de décisions par le manager. Les résultats doivent être calculés en utilisant le multithreading avec des contraintes temporelles fortes pour accommoder la sensibilité et la criticité de l'écosystème. Les données de configuration de l'application telles que le nombre de threads/processus à lancer, la fréquence d'interrogation des capteurs, et les mécanismes de synchronisation/communication à utiliser doivent être spécifiées par un fichier XML ou par des arguments passés à l'application au moment de l'exécution.

Pour commencer, je vous donnerai un code qui constitue une solution partielle qu'on peut améliorer successivement pour obtenir une solution de qualité satisfaisant l'ensemble de besoins du cahier des charges.

Travail à faire

1) Elaborer une architecture logique (logicielle) et une autre physique (d'exécution) de l'application tout en indiquant le rôle de chaque entité incluse dans votre solution.
2) Exécuter le code fourni en faisant changer le nombre de threads à utiliser tout en calculant à chaque fois le temps d'exécution de l'application et le pourcentage d'utilisation de temps du processeur.
3) Changer le code de l'application en remplaçant les threads par des processus lourds crées à l'aide des appels système `fork()`. Comparer cette solution avec la solution précédente en termes de temps d'exécution, temps processeur, ressources mémoire, simplicité, et évolutivité.
4) Reprendre la solution fournie en changeant le code des threads utilisés pour effectuer le calcul de la somme des températures en utilisant une seule zone mémoire pour tous les threads au lieu d'une zone spécifique par thread.
5) Procéder au "refactoring" du code de la solution pour une qualité meilleure en termes de modularité, configurabilité, simplicité, maintenabilité, et réutilisabilité. Pour ce faire, je vous demande de décomposer l'application en des modules dont chacun se charge d'un aspect de l'application (lecture, écriture, calcul, coordination, configuration, etc.) en adoptant le principe de "séparation de préoccupations/responsabilités" ou "separation of concerns en anglais". Chaque module doit consister en un fichier ".h" et un fichier ".c", et un fichier Makefile est souhaitable pour faciliter la compilation et l'édition de liens des modules résultats.
6) Reprendre la solution fournie en changeant la logique-métier pour permettre de calculer les statistiques demandées en affectant des poids différents pour les températures collectées. Pour ce faire, on a besoin d'intégrer une matrice (`n x n`) de poids puis de calculer le produit des deux matrices (`matrice_températures x matrice_poids`) avant de calculer la somme pondérée des températures. Pour bien faire, on a besoin de paralléliser le calcul de produits en utilisant le multithreading.
7) Elaborer un compte rendu de ce TP par équipe de trois étudiants et de quatre pages au maximum à rendre au plus tard **le dimanche 30 novembre 2020 à minuit**.

Bon apprentissage !