Rapport ENSTA 2022 – ROB305 – Juan Nicolas MORENO DIMATE

[TD-1] Mesure de temps et échantillonnage en temps

Cette section présente le travail effectué pour le TP1 du cours ROB305, la base de code Du TP se trouve dans le dossier TP1/src, où chaque 'main_tdx.cpp' correspond à l'un des sections de TP.

a) Gestion simplifiée du temps Posix

Dans la première instance de cette section, on a implémenté des fonctions et des opérateurs qui permet d'utiliser la structure *timespect*, qui fournit la mesure du temps dans l'API Posix. Ces fonctions sont définies dans le *dossier timerLib*, dans le fichier *timerLib.h*, et sont implémentées dans le fichier *timerLib.cpp*.

Il faut noter que la structure timespec présente une mesure de temps composée en valeurs en secondes (tv_sec) et en valeurs en nanosecondes (tv_nsec), où la valeur en nanosecondes est strictement supérieure ou égale à zéro. La fonction main_td1a.cpp présente plusieurs tests pour tester les fonctions et opérateurs définis.

b) Timers avec callback

Pour cette section, l'objectif était d'implémenter un compteur *Posix* périodique avec une fréquence de 2 Hz, qui effectue un incrément périodique et imprime la valeur de l'incrément. Le compteur effectue le processus jusqu'à 15 incréments, puis il est terminé. Pour cela, on a utilisé les signaux *sigevent* et *sigitacion*, qui sont responsables du contrôle de l'expiration du *timer*. Pour cette raison, l'utilisation de callbacks est nécessaire.

Afin de rendre l'implémentation plus modulaire et mieux distribuée, le processus a été divisé en 2 fonctions, une responsable de la création des timers et une fonction *'handler'* permettant l'utilisation du callback. Ainsi, après l'expiration d'un signal d'événement, l'action est déclenchée et le compteur est incrémenté.

```
void myHandler(int, siginfo_t* si, void*)
{
    volatile int& count_ref = *((int*) si->si_value.sival_ptr);
    std::cout<<"My Counter :" << count_ref << std::endl;
    count_ref += 1;

}
```

c) Fonction simple consommant du CPU

Pour le développement de cette sous-section, une fonction *incr* implémentant un compteur a été programmée, avec pour fonction d'effectuer *nLoops* fois la boucle qui incrémente le compteur pointé par le pointeur *pCounter*, comme indiqué ci-dessous :

```
void incr(unsigned int nLoops, double* pCounter){
    unsigned int i = 0 ;
    while(i<nLoops){
        *pCounter += 1.0;
        i++;
        }
}
```

La fonction principale dans ce cas a été construite avec une signature standard sous la forme *int main(int argc, char* argv[])*, qui permet la vérification des arguments insérés et la récupération de ces valeurs à travers le vecteur *argv[]*, où le premier argument est indexé comme *argv[1]*.

Dans ce cas, il est utile de préciser qu'il est nécessaire de fournir un argument dans l'exécution binaire, donc une section de vérification est ajoutée dans le code au cas où une exécution est effectuée sans l'argument requis, afin d'éviter les échecs de segmentation. Cette même section sera mise en œuvre à des points ultérieurs où l'entrée de l'utilisateur est requise.

d) Mesure du temps d'exécution d'une fonction

Dans cette section, on utilise la fonction *incr*() définie précédemment, en ajoutant un argument supplémentaire *pStop* de type booléen, qui permet de limiter l'incrémentation du compteur en fonction de son état. Etant donné que la valeur de cette variable peut être modifiable à tout moment, une fonction *myHandler*() est implémentée spécialement dédiée à la modification de ce paramètre.

```
void myHandler(int, siginfo_t* si, void*){
 *((bool*)si->si_value.sival_ptr) = true;
}
```

Une fonction *calib()* est également codée pour permettre la calibration des coefficients d'une fonction linéaire de la forme $l(t) = a * x + b$, en utilisant la fonction *setIncrement(time_t sec)* avec différentes valeurs comme arguments, ce qui permet d'obtenir une approximation directe des paramètres a et b, comme le montre la fonction présentée ci-dessous :

```
coef calib()
{
    coef params;
    cout << "Calibration " << endl;
    double iLoop_4 = setIncrement((time_t) 4);
    double iLoop_6 = setIncrement((time_t) 6);

    // Estimating the coefficients
    params.a = (iLoop_6 - iLoop_4)/(2);
    params.b = (iLoop_6 - params.a*6);

    return params;
}
```

Les modifications apportées peuvent être testées dans le fichier *main_td1d.cpp* du dossier *src*, où les coefficients sont déterminés en comparant le nombre de boucles utilisées par un appel de la fonction setIncrement par rapport à la fonction de calibration.

[TD-2] Familiarisation avec l'API multitâches *pthread*

a) Exécution sur plusieurs tâches sans mutex

Dans cette section on a mis en place la parallélisation du processus de comptage à travers des *threads* afin de réaliser une variable de comptage incrémentale. Idéalement, chaque thread déclaré devrait pouvoir effectuer le nombre d'itérations spécifié et ainsi obtenir comme résultat final du compteur une valeur égale au nombre d'itérations spécifié par le nombre de threads déclarés.

Dans ce cas, une *structure threadData* a été définie pour permettre l'utilisation de fils dans la fonction *incr* précédemment définie. Pour cela, une fonction de la forme void* call_incr(void* threadData) a été créée afin d'effectuer l'appel à incr. La fonction présentée ici ainsi que la structure créée sont utilisées dans la fonction pthread_create() pour lancer les *nTask* threads, via la fonction pthread_join().

Les tests effectués montrent que la valeur du compteur atteint toujours une valeur inférieure à celle attendue, ceci est dû au fait que les trets modifient la valeur du compteur en même temps, ce qui entraîne des problèmes d'accès parallèle aux mêmes données (écriture concurrente), ce qui affecte finalement le résultat attendu. La solution à ce problème est l'implémentation du Mutex, comme nous le verrons dans les sections suivantes.

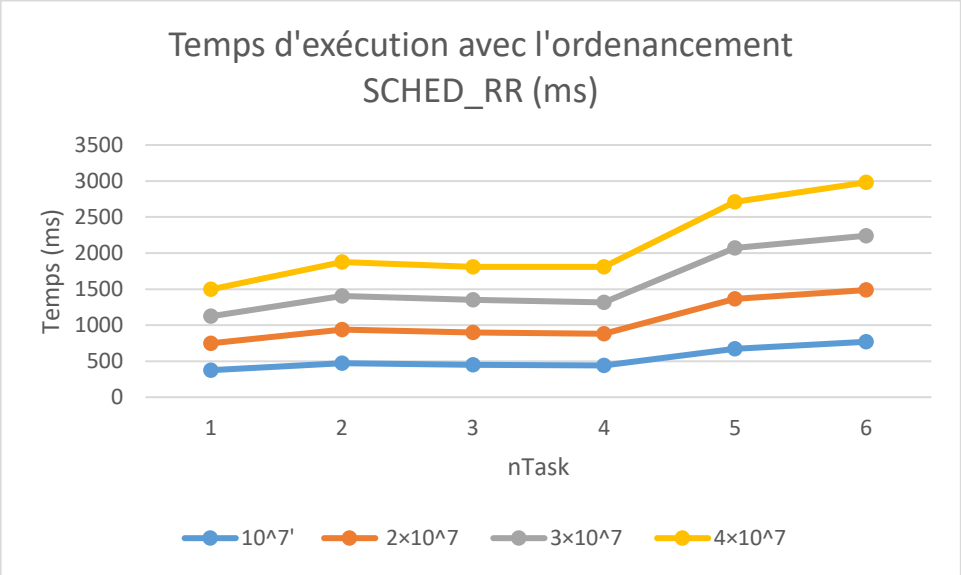
b) Mesure de temps d'exécution

Pour cette étape, une modification a été apportée à la base de code qui permet d'attacher une politique de tri (comme argument de la fonction principale, argv[3]), en considérant les politiques SCHED_RR, SCHED_FIFO et SCHED_OTHER. La politique utilisée par défaut est SCHED_OTHER, qui donne une priorité de 0. Les 2 autres politiques donnent une priorité maximale.

Le programme du fichier *main_td2b.cpp* a été exécuté de nombreuses fois avec une politique d'ordonnancement fixe (SCHED_RR), en faisant varier les paramètres du nombre de cycles d'exécution nLoops ($1 \times 10\hat{}7$, $2 \times 10\hat{}7$, $3 \times 10\hat{}7$, $4 \times 10\hat{}7$) et du nombre de tâches, nTask (1, 2, 3, 4, 5, 6). Les résultats obtenus à partir du balayage paramétrique sont présentés ci-dessous :

Temps d'exécution avec l'ordonnancement SCHED_RR (ms)				
nTask\nLoops	**10^7**	**2×10^7**	**3×10^7**	**4×10^7**
1	374.938	748.344	1122.81	1496.38
2	470.008	938.229	1406.5	1875.74
3	450.615	900.083	1349.33	1810.21
4	439.406	878.201	1316.87	1809.07
5	671.593	1364.75	2070.15	2710.91
6	768.815	1487.2	2240.98	2980.99

Comme il est évident, plus le nombre de cycles d'exécution augmente, plus le temps d'exécution global du programme augmente également, et plus de nouvelles tâches sont ajoutées, plus le temps d'exécution augmente également, mais dans ce cas, l'augmentation n'est pas linéaire. Le graphique ci-dessous montre un saut dans le temps pris pour passer de la quatrième ou cinquième tâche, ce qui signifie que l'architecture du processeur du Raspberry utilisé a quatre threads actifs.



c) Exécution sur plusieurs tâches avec mutex

Dans cette section, un autre argument de protection est ajouté au code programmé dans la première section du TD2. Dans ce cas, la structure précédemment programmée est modifiée en ajoutant un Mutex et une variable booléenne pour sa mise en œuvre. De même, la fonction d'appel du Handler est modifiée pour l'implémentation du mutex comme indiqué ci-dessous :

```
struct Data
{
  int nLoops;
  double pCounter;
  bool isProtected;
  pthread_mutex_t mtx;
};
```

Et pour la fonction d'appel du Handler :

```
void* call_incr(void* vThreadData)
{
    Data* pThreadData = (Data*) vThreadData;

  if (pThreadData->isProtected)
  {
    pthread_mutex_lock(&pThreadData->mtx);
    incr(pThreadData->nLoops, (double*) &pThreadData->pCounter);
    pthread_mutex_unlock(&pThreadData->mtx);
  }
  else
  {
    incr(pThreadData->nLoops, (double*) &pThreadData->pCounter);
  }
    return vThreadData;
}
```

Les tests de la modification effectuée sont présentés dans le fichier main_td2c.cpp, où l'on peut voir que grâce à l'implémentation du Mutex, les erreurs d'écriture simultanée sont corrigées, de sorte que la valeur attendue du compteur est obtenue.

[TD-3] Classes pour la gestion du temps

a) Classe Chrono

Dans cette section, on a implémenté la classe '*Chrono*' qui permet d'implémenter des fonctions de mesure du temps. À l'aide de cette classe, il est possible de définir le début, la fin et la durée d'un processus, et il existe également des fonctions qui permettent d'utiliser facilement l'outil, comme une méthode de réinitialisation et une méthode de contrôle de l'état du chronomètre.

Le fichier *main_td3a.cpp* présente les tests effectués sur les méthodes définies dans la classe Chrono.

b) Classe Timer

Dans cette section, une classe Timer a été implémentée, qui permet d'encapsuler les fonctions d'un timer Posix.

Dans ce cas, le constructeur, le destructeur, la méthode start() et la méthode stop() sont déclarés comme publics car ils doivent être appelés dans le main. Le constructeur a été réalisé avec les structures sigaction et sigevent. La méthode callback() est déclarée comme une variable protégée car il s'agit d'une variable virtuelle qui sera implémentée par une classe enfant qui hérite de Timer, mais pas par Timer.

La méthode call_callback() est déclarée comme une méthode privée car elle ne sera utilisée que par la classe Timer, et est déclarée comme une variable statique car elle ne peut être instanciée que par des objets de cette même classe.

La classe CounterDown() hérite de la classe Periodic(), qui elle-même hérite de la classe Timer. La classe CounterDown() permet l'implémentation directe d'un compteur avec une fréquence de 1Hz, qui décrémente d'un nombre arbitraire fourni jusqu'à zéro.

Le fichier main_td3b.cpp contient une fonction principale qui instancie un objet CounterDown, qui décrémente une valeur *nLoops* définie par l'utilisateur.

c) Calibration en temps d'une boucle

Dans cette section et sur la base du TD1 partie d, on a cherché à encapsuler les fonctions qui réalisent l'estimation des paramètres a et b d'une fonction de type l(t)= a*t+b. Pour cela, on a créé une classe Calibrator() dérivée de PeriodicTimer, qui prend les valeurs d'une instance de la classe Looper, et qui est instanciée dans une classe CpuLoop, qui permet de démarrer le compteur en l'incrémentant jusqu'à un nombre d'échantillons déterminé par l'utilisateur avec lequel on réalisera l'estimation des paramètres. Le fichier main_td3c.cpp présente les tests permettant de vérifier le processus de calibration.

[TD-4] Classes de base pour la programmation multitâche

Cette section a examiné l'encapsulation de la gestion des tâches Posix à travers l'implémentation des classes PosixThread, Mutex et Lock.

a) Classe Thread

Initialement, pour cette section, la classe PosixThread a été implémentée, où les paramètres de base pour l'implémentation des threads sont définis, y compris la politique d'ordonnancement à suivre. La classe Thread en question est alors un dérivé de la classe PosixThread. Afin de vérifier le bon fonctionnement des classes mentionnées, une classe supplémentaire CounterThread a été créée, pour l'implémentation d'un compteur tel que celui observé dans le TD2 partie a.

Les tests effectués dans ce cas ont révélé le même problème d'écriture concurrente que celui de Td 2 partie a. Par conséquent, la définition et l'implémentation d'une classe Mutex sont proposées pour résoudre ce conflit.

b) Classes Mutex et Mutex::Lock

Pour cette section, une classe Mutex() a été conçue, ainsi que les classes héritées Lock(), TryLock() et Monitor(), afin d'effectuer la gestion du Mutex dans l'application. Les classes Lock() et TryLock() permettent d'obtenir et de libérer le Mutex. La classe Monitor() est chargée de gérer l'accès des Threads au Mutex par le biais des fonctions notify() et wait().

Dans ce cas et afin de faciliter l'implémentation du Mutex, une classe CounterMutex héritée de Thread a également été créée, qui implémente une méthode run() qui active le compteur, cette fois en présence d'un Mutex, qui instancie la classe Mutex::Lock avant l'exécution de l'incrément, résolvant ainsi le problème de l'écriture concurrente. Les tests effectués pour tester les classes et méthodes implémentées se trouvent dans le fichier main_td4b.cpp.

c) Classe Semaphore

La classe Semaphore a été créée afin d'encapsuler les fonctionnalités d'un *semaphore* en prenant et en donnant des jetons. Dans cette classe est définie la méthode give() qui gère la livraison des jetons, la méthode take() par laquelle les jetons sont retirés, et la surcharge de la méthode take() pour l'appel à une instance de Mutex::Lock et l'affectation d'un délai d'attente dans la fonction déjà implémentée.

De même, des classes dérivées de Thread ont été créées pour permettre, par le biais de pointeurs, l'utilisation du même sémaphore dans le main, et son utilisation par la production et la consommation de jetons à l'aide de la méthode run(). Dans ce cas, pour

faciliter l'implémentation, on définit une classe SemaphoreProducer() et une classe SemaphoreConsumer() dont les objectifs sont de donner des jetons au sémaphore par la méthode give(), ou de prendre des jetons du sémaphore par la méthode take(). Les tests effectués pour tester les classes et méthodes implicites se trouvent dans le fichier main_td4c.cpp, à l'intérieur du fichier on peut voir les tests des classes producer et consumer à travers l'utilisation d'un sémaphore partagé. Les paramètres du nombre de producteurs (nProd), du nombre de consommateurs (nCons) et du nombre d'articles (nItems) sont donnés par l'utilisateur au programme.

d) Classe Fifo multitâches

Dans cette section, un modèle Fifo est créé en utilisant *std::queue* dans un fichier .hpp. Dans cette classe, les méthodes pop() et push() sont créées pour implémenter le système *First Input First Output*.

La méthode pop() et sa surcharge par l'inclusion d'un paramètre de temps utilisent les fonctions pop() et front() pour prendre le premier élément, en utilisant à tout moment un Mutex() pour protéger l'opération. Dans ce cas, comme dans les sections précédentes, les classes FifoConsumer et FifoProducer ont également été créées pour l'implémentation et le test de la classe Fifo. Globalement, la classe FifoProducer, par ses méthodes, produit une série de nombres de zéro jusqu'à une limite donnée, qui sont consommés par la classe FifoConsumer.

Le fichier main_td4d.cpp présente la fonction *main()* qui teste la fonctionnalité de la classe Fifo. L'algorithme implémenté pour le test du Sémaphore est utile dans ce cas pour la division des nombres à consommer par les Tâches. Le contrôle implémenté effectue une vérification que le nombre d'éléments produits est égal au nombre d'éléments consommés.