Réseaux

Julien Jandot 10/01/2018

Les threads

Les threads sont des fils d'exécution qui permettent d'exécuter plusieurs suites d'instructions en parallèles, à l'intérieur du **même** processus. Chaque thread a sa pile et ses variables, et peut en partager certaines avec d'autres threads.

Thread simple avec paramètre

```
1  #include <iostream>
2  #include <pthread.h>
3  #include <cstdlib>
4
5  void *fonctionThread(void *param){
6      int numero = (int) param;
7      std::cout << "Je suis le thread : " << numero << std::endl;
8      pthread_exit(nullptr);
9  }
10
11 int main(){
12     pthread_t idThread;
13     int numero = 1643;
14     if(pthread_create(&idThread, nullptr, fonctionThread, (void *) numero) != 0){
15         std::cerr << "Erreur lors de la creation" << std::endl;
16         std::exit(-1);
17     }
18
19     if(pthread_join(idThread, nullptr) != 0){
20         std::cerr << "Erreur lors de la fermeture de thread" << std::endl;
21         std::exit(-1);
22     }
23
24     return 0;
25 }
```

Les mutex

Pour éviter que des ressources partagées soient modifées simultanément par plusieurs threads, ce qui peut entrainer des interblocages, il existe un système de verrous, **les mutex**.
Les mutex ont deux états, libre et occupé. Seul un thread peut verrouiller un mutex, l'opération de verrouillage est bloquante. Seul le thread verrouillant peut déverrouiller le mutex, qui pourra donc de nouveau être utlisé par les autres threads.
Il existe deux manières d'initialiser un mutex, de manière 'classique' et de manière statique. La déclaration dans tous les cas se fait dans un segment commun à tous les threads, donc généralement comme une variable globale. Aujourd'hui l'initialisation statique est préférée. [1]

Utilisation de mutex

```cpp
#include <iostream>
#include <sys/types.h>
#include <pthread.h>

//INITIALISATION STATIQUE
pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;

//INITIALISATION DYNAMIQUE
pthread_mutex_t verrou;

void *fonctionThread(void * param){
    pthread_mutex_lock(&verrou);
    std::cout << "Thread : verrou obtenu" << std::endl;
    /*
        OPERATION CRITIQUE
    */
    pthread_mutex_unlock(&verrou);
    std::cout << "Thread : verrou rendu" << std::endl;
    pthread_exit();
}

int main(){
    //INITIALISATION DYNAMIQUE SUITE
    if(pthread_mutex_init(&verrou, nullptr) != 0){
        std::cerr << "Erreur lors de l'initialisation du mutex" << std::endl;
        std::exit(-1);
    }
    pthread_t idThread;
    if(pthread_create(&idThread, nullptr, fonctionThread, nullptr) != 0){
        std::cerr << "Erreur lors de la creation" << std::endl;
        std::exit(-1);
    }
```

1. https://stackoverflow.com/questions/14320041/

```
33
34      pthread_mutex_lock(&verrou);
35      std::cout << "Main : verrou obtenu" << std::endl;
36      /*
37          OPERATION CRITIQUE
38      */
39      pthread_mutex_unlock(&verrou);
40      std::cout << "Main : verrou rendu" << std::endl;
41
42      if(pthread_join(idThread, nullptr) != 0){
43          std::cerr << "Erreur lors de la fermeture de thread" << std::endl;
44          std::exit(-1);
45      }
46
47      return 0;
48  }
```

Les variables conditionnelles

On a parfois besoin de connaître l'état d'une donnée en temps réel, pour pouvoir faire des traitements en fonction de cette donnée. Les mutex offrent une solution à ce problème, mais elle est inefficace.
Pour subvenir à ce besoin, il faut utiliser les **variables conditionnelles**.
Une **variable conditionnelle** est une donnée commune à plusieurs threads, fonctionnant comme un booléen et symbolisant l'occurrence d'un évènement, autrement dit, la satisfaction d'un prédicat, à la seule différence que la condition est bloquante (si la variable conditionnelle n'est pas validée, le thread sera en attente jusqu'à sa validation).

Variables conditionnelles sur deux threads

```
1  #include <iostream>
2  #include <pthread.h>
3  #include <cstdlib>
4
5  //INITIALISATION STATIQUE
6  pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER;
7  pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
8  int x;
9
10
11 void *fonction1(void * param){
12     std::cout << "Fonction 1 : start" << std::endl;
13     for(int i = 0; i < 20; ++i){
14         pthread_mutex_lock(&verrou);
15         ++x;
16         std::cout << x << std::endl;
17         if(x == 10) {
18             std::cout << "Signal" << std::endl;
19             pthread_cond_broadcast(&cond);
20         }
21         pthread_mutex_unlock(&verrou);
22     }
23     std::cout << "Fonction 1 : end" << std::endl;
24     pthread_exit(nullptr);
25 }
26
27 void *fonction2(void * param){
28     std::cout << "Fonction 2 : start" << std::endl;
29     pthread_mutex_lock(&verrou);
30     while(x < 10){
31         pthread_cond_wait(&cond, &verrou);
32     }
33     x += 10;
34     pthread_mutex_unlock(&verrou);
```

```
35      std::cout << "Fonction 2 : end" << std::endl;
36      pthread_exit(nullptr);
37  }
38
39  int main(){
40      pthread_t idThread1;
41      pthread_t idThread2;
42      if(pthread_create(&idThread1, nullptr, fonction1, nullptr) != 0){
43          std::cerr << "Erreur lors de la creation de thread 1" << std::endl;
44          std::exit(-1);
45      }
46      if(pthread_create(&idThread2, nullptr, fonction2, nullptr) != 0){
47          std::cerr << "Erreur lors de la creation de thread 2" << std::endl;
48          std::exit(-1);
49      }
50      if(pthread_join(idThread1, nullptr) != 0){
51          std::cerr << "Erreur lors de la fermeture de thread 1" << std::endl;
52          std::exit(-1);
53      }
54      if(pthread_join(idThread2, nullptr) != 0){
55          std::cerr << "Erreur lors de la fermeture de thread 2" << std::endl;
56          std::exit(-1);
57      }
58      return 0;
59  }
```

À la ligne 19, la fonction **pthread_cond_broadcast(pthread_cond_t *condition)** est appelée. Cette fonction va notifier les autres threads en attente que la condition, et les réveiller.
Une autre fonction est utilisable, il s'agit de **pthread_cond_signal(pthread_cond_t *condition)**. Alors que la première réveille tous les threads en attente, celle ci va notifier un seul thread et le réveiller. L'utilisateur ne peut pas choisir quel thread réveiller, l'utilisation de pthread_cond_signal() ne doit donc se faire que lorsque tous les threads peuvent effectuer l'action requise.
En cas de doute, ou si la variable conditionnelle est utilisée par plusieurs prédicats, **pthread_cond_broadcast()** est préférable.

Les IPC

Il existe trois types d'objet IPC : les files de message, la mémoire partagée et les ensembles de sémaphores. Ces objets peuvent être publics ou privés. Pour accéder à un objet public, il faut un identifiant, une clé. Cette clé est obtenue de manière identique pour tous les objets IPC, grâce à la fonction suivante :

key_t ftok(const char* path, int id)

La paramètre path est le chemin vers le fichier commun, et id correspond à un entier quelconque, choisi préalablement.

Les files de message

Une file de messages est une structure en memoire centrale, faite pour communiquer entre processus, l'unite d'echange étant un message.
Chaque processus peut créer, utiliser une file dont il a la clé, détruire une file. L'utilisation est la suivante : déposer ou extraire un message.
Une file de message est indépendante du processus qui l'a créée, il faut donc penser à la détruire à la fin du programme, sinon elle conservera son contenu jusqu'à un redémarrage système.
Si la file est pleine ou vide, les processus voulant respectivement déposer ou retirer un message seront mis en attente.

File de message

```
1  #include <csignal>
2  #include <cstring>
3  #include <iostream>
4  #include <pthread.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/msg.h>
8
9  typedef struct Message Message;
10 struct Message{
11     long etiquette;
12     char message[128];
13 };
14 int idFile;
15
16 void sig_handler(int signal){
17     if(signal == SIGINT){
18         //destruction de la file de message
19         msgctl(idFile, IPC_RMID, NULL);
20         exit(EXIT_SUCCESS);
21     }
22 }
23
```

```
24
25 int main(){
26     if(signal(SIGINT,sig_handler) == SIG_ERR){
27         std::cerr << "Erreur SIGINT" << std::endl;
28         exit(EXIT_FAILURE);
29     }
30
31     //CREATION DU TOKEN COMMUN CLIENT SERVEUR
32     key_t token = ftok("fichier", 1);
33
34     idFile = -1;
35     idFile = msgget(token, IPC_CREAT | 0666);
36     //si -1 erreur
37
38     Message m;
39
40     while(1){
41         //attente de message
42         if(msgrcv(idFile, &m, (size_t)sizeof(Message), 1, 0) == -1){
43             //traitement erreur sig_handler(SIGINT)
44         }
45
46         /*
47             traitement du message
48         */
49
50         //envoi de message dans la file
51         if(msgsnd(idFile, &m, strlen(m.message)+1, 0) == -1){
52             //traitement erreur sig_handler(SIGINT)
53         }
54     }
55     exit(EXIT_FAILURE);
56 }
```

Fonctionnement des étiquettes et de msgrcv

Si le fonctionnement de msgsnd est classique, on remarque un paramètre de plus dans msgrcv. Le flag a 1 dans le code, ligne 41, correspond à l'étiquette. C'est cette étiquette qui permet de retirer un message en particulier. En effet, chaque message, en plus de son contenu, à une étiquette, qui doit être le premier paramètre de la struct. La valeur de l'etiquette est importante, on va pouvoir l'utiliser pour adresser des messages uniquement à certains processus. On note trois cas possibles :

- etiquette > 0 : on va lire le premier message qui a la meme valeur d'étiquette
- etiquette = 0 : on va lire le premier message disponible dans la file
- etiquette < 0 : on va lire le premier message dont l'étiquette est la plus petite parmi celles dont la valeur

est inférieure à la valeur absolue de l’étiquette

La mémoire partagée

De manière générale, chaque processus a sa propre mémoire, inaccessible depuis un autre processus. La mémoire partagée est une solution à ce problème, en fournissant un espace commun à plusieurs processus. On peut ajouter des restrictions suivant le processus, comme le droit de lecture ou d'écriture. Comme pour la file de messages, le segment de mémoire partagée n'appartient pas à un processus unique, il faut donc penser à le détruire à la fin de l'utilisation.
Les opérations disponibles sont les suivantes : création, attachement, détachement, et suppression. Les droits sont donnés selon le même principe que pour les fichiers.

Mémoire partagée

```
1  typedef struct Parking{
2      int nbPlace;
3  } Parking;
4
5  int shmid;
6
7  void sig_handler(int signal){
8      if(signal == SIGINT){
9          //destruction du segment
10         shmctl(shmid, IPC_RMID, NULL);
11         exit(EXIT_SUCCESS);
12     }
13 }
14
15 int main(){
16     if(signal(SIGINT,sig_handler) == SIG_ERR){
17          std::cerr << "Erreur SIGINT" << std::endl;
18          exit(EXIT_FAILURE);
19     }
20     key_t key = ftok("bla", 1);
21
22     Parking parking;
23     Parking *p_parking;
24     p_parking = &parking;
25     (*p_parking).nbPlace = 5;
26
27     /*
28      * Creer le segment
29      */
30     shmid = -1;
31     if ((shmid = shmget(key, sizeof(int), IPC_CREAT | 0666)) < 0) {
32         perror("shmget");
33         exit(1);
34     }
35
```

```c
    /*
     * Lier le segment
     */
    if ((p_parking = shmat(shmid, NULL, 0)) == (Parking *) -1) {
        perror("shmat");
        exit(1);
    }


    /* Traitement */
    (*p_parking).nbPlace = 5;
     *p_parking = parking;
    while((*p_parking).nbPlace > 0){
        sleep(1);
    }
    /* Fin traitement */

    //detachement
    int dtres = shmdt((void*)p_parking);
    if(dtres < 0)
        printf("Erreur dtres\n");

    //supression
    int destroy = shmctl(shmid, IPC_RMID, NULL);
    if(destroy < 0)
        printf("Erreur destroy\n");
    return 0;
}
```

Les ensembles de sémaphore

Un sémaphore est un mécanisme de synchronisation de processus. Il s'agit d'une structure de données qui comprend : un entier non négatif donnant le nombre de ressources disponibles, une file d'attente de processus. Cette structure est manipulée au travers de plusieurs opérations : **Init (sémaphore sem, int nbRessources)**, **P(sémaphore sem, int nbRessources)** qui bloque l'appelant si le nombre de ressources de sem demandées est supérieur au nombre de ressources disponibles, sinon décrémente le nombre de ressources de sem et **V(sémaphore sem, int nbRessources)** qui libère un nombre de ressources obtenues et débloque un ou des processus en attente s'il en existe. Une opération d'attente Z existe et sera détaillée.

Certaines opérations se doivent d'être plus détaillées qu'une simple portion de code.
Pour réaliser un ensemble d'opération sur un ensemble de sémaphores, on utilise la fonction

int semop(int idSem, struct sembuf *tabOp, int nbOp)

Le premier paramètre correspond au résultat de semget, de manière similaire aux autres IPC. Le dernier paramètre correspond au nombre d'opération présentes dans la struct tabOp, globalement la taille du tableau. La struct sembuf comporte plusieurs paramètres :

Détail de sembuf + exemple

```
1  struct sembuf{
2      unsigned short sem_num; //numero du semaphore ou s'executera l'action
3      short sem_op;
4      short sem_flg; //options, generalement 0, ou SEM_UNDO
5  }
6
7
8  struct sembuf sops[2];
9  int semid;
10
11 key_t key = ftok("bla", 1);
12
13 semid = semget (key, 2, 0666);
14
15 sops[0].sem_num = 0;        /* Agir sur le semaphore 0 */
16 sops[0].sem_op = 0;         /* Attendre que la valeur soit de 0 */
17 sops[0].sem_flg = 0;
18
19 sops[1].sem_num = 0;        /* Agir sur le semaphore 0 */
20 sops[1].sem_op = 1;         /* Incrementer la valeur de 1 */
21 sops[1].sem_flg = 0;
22
23 if(semop(semid, &sop, 2) == -1) {
24     std::cerr << "Erreur op" << std::endl;
25     exit(EXIT_FAILURE);
26 }
```

Le second paramètre sem_op, va définir l'opération exécutée, avec trois valeurs N possibles :

- N > 0 : opération V, on incrémente le nombre de ressources disponible de N, et on réveille les processus en attente.
- N < 0 : opération P, on essaye de retirer | N | ressources
- N = 0 : opération Z, on attends que la valeur du sémaphore soit à 0

Si une opération comporte le flag SEM_UNDO, elle sera annulée à la fin du processus.

Pour réaliser des opérations dont l'initialisation de sémaphores, on va utiliser la fonction

int semctl(int semid, int semnum, int cmd, union semun arg)

Le dernier paramètre n'est pas obligatoire, il va dépendre de la valeur de cmd. Ici, les opérations SETVAL ET GETVAL vont être vues, la liste complète est disponible ici [2]

Détail de l'union semun + exemple

```
1  struct semun{
2      int val; //obligatoire si cmd = SETVAL
3      struct semid ds *buf;
4      unsigned short *array;
5      struct seminfo *buf; //utilisable uniquement sous linux
6  }
7
8  semun arg;
9  arg.val = 1;
10 if(semctl(semid, 0, SETVAL, arg) == -1){ //La valeur des semaphores sera initialisee ↩
       a 1
11     std::cerr << "Erreur init" << std::endl;
12 }
13
14 int semValue = semctl(semId, 0, GETVAL);
15 //semValue prends la valeur du nombre de ressources disponibles du semaphore
```

Pour finir, la destruction se fait de la manière classique suivante : **semclt(idSem, 0, IPC_RMID)**

2. http://manpagesfr.free.fr/man/man2/semctl.2.html

Les sockets

Une socket est une notion qui étend celle de tube. Elle permet de définir un canal de communication entre deux processus, sauf qu'elle permet aussi la communication en réseaux, et le choix des protocoles de communication.

Une socket est définie par :

- un domaine, généralement PF_INET, pour la communication IPv4
- un type, SOCK_STREAM(TCP) ou SOCK_DGRAM(UDP)
- une adresse IP, et un numéro de port

Les différences entre le mode TCP et UDP sont les suivantes :

TCP	UDP
fiable	non fiable
ordre garanti	ordre non garanti
duplication impossible	duplication possible
mode connecté(ex : téléphone)	mode non connecté(ex : poste)

Opérations de base des sockets TCP et UDP

```
1  int sock = socket(PF_INET, SOCK_DGRAM, 0); //ou SOCK_STREAM si on utilise TCP
2  if(sock == -1){
3      std::cerr << "Erreur socket" << std::endl;
4      return sock;
5  }
6
7  struct sockaddr_in in;
8  in.sin_family = AF_INET;
9  in.sin_addr.s_addr = INADDR_ANY;
10 in.sin_port = htons(10080);
11
12 if(bind(sock, (struct sockaddr*)&in, sizeof(in)) < 0){
13     std::cerr << "Erreur bind" << std::endl;
14     return -1;
15 }
16
17 /*
18     Traitement particulier a TCP ou UDP
19 */
20
21 if(close(sock) == -1){
22     std::cerr << "Erreur close" << std::endl;
23     std::exit(-1);
24 }
```

Code UDP

Serveur UDP

```
int sock = socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr in addr;
addr.sin_family = AF INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(atoi(argv[1]));
bind(dS, (struct sockaddr*)&addr, sizeof(addr));
struct sockaddr_in addrExp;
socklen t sizeExp = sizeof(struct sockaddr_in);
char m [20];
recvfrom(sock, m, sizeof(m), 0, &addrExp, &sizeExp);
std::cout << "Recu : " << m << std::endl;
int r = 10;
sendto(sock, &r, sizeof(int), 0, (sockaddr*)addrExp, sizeExp);
close(sock);
```

Client UDP

```
int sock = socket(PF_INET, SOCK_DGRAM, 0);
struct sockaddr_in addrDest;
addrDest.sin family = AF_INET;
inet_pton(AF_INET, argv[1], &(addrDest.sin_addr));
addrDest.sin_port = htons(atoi(argv[2]));
socklen_t sizeAddr = sizeof(struct sockaddr_in);
char *m = "Bonjour";
sendto(sock, m, 8, 0, (sockaddr*)addrDest, sizeAddr);
int r;
recvfrom(sock, &r, sizeof(int), 0, NULL, NULL);
std::cout << "r" << std::endl;
close (sock);
```

Code TCP

Serveur TCP

```
1  int sock = socket(PF_INET, SOCK_STREAM, 0);
2  struct sockaddr_in ad;
3  ad.sin_family = AF_INET;
4  ad.sin_addr.s_addr = INADDR_ANY;
5  ad.sin_port = htons(atoi(argv[1]));
6  bind(sock, (struct sockaddr*)&ad, sizeof(ad));
7  listen(sock, 7);
8  struct sockaddr_in addrClient;
9  socklen_t sizeClient = sizeof(struct sockaddr_in);
10 int sockClient = accept(sock, (struct sockaddr*) &addrClient,&sizeClient);
11 char msg [20];
12 recv(sockClient, msg, sizeof(msg), 0);
13 std::cout << msg << std::endl;
14 int r = 10;
15 send(sockClient, &r, sizeof(int), 0);
16 close(sockClient);
17 close(sock);
```

Client TCP

```
1  int sock = socket(PF_INET, SOCK_STREAM, 0);
2  struct sockaddr_in addrServeur;
3  addrServeur.sin_family = AF_INET;
4  inet_pton(AF INET,argv[1],&(addrServeur.sin_addr));
5  addrServeur.sin_port = htons(atoi(argv[2]));
6  socklen_t sizeAddr = sizeof(struct sockaddr_in);
7  connect(sock, (struct sockaddr *) &addrServeur, sizeAddr);
8  char *m = "Bonjour";
9  send(sock, m, 8, 0);
10 int r;
11 recv(sock, &r, sizeof(int), 0);
12 std::cout << r << std::endl;
13 close (sock);
```