UNIVERSITÉ DE ROUEN

UFR Sciences et Techniques

Rapport de Projet

Thème : réalisation d'un lanceur de commandes

Manuel Technique

Réaliser par :

EL KOUAY El Mehdi

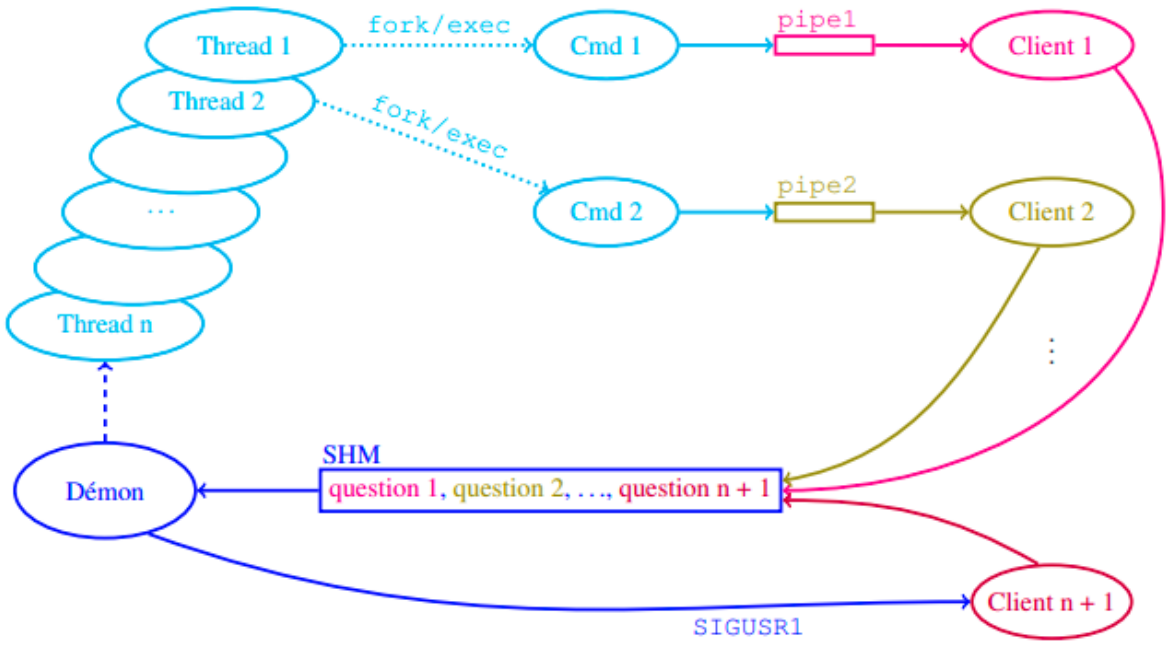
BERHAIL Khalid

Licence 3 Informatique | 2020 - 2021

I. Description générale du projet.

Le but de ce projet est de réaliser un lanceur de commandes qui permet à plusieurs clients de rentrer des commandes à exécuter par un serveur, et que les résultats de cette exécution sera envoyer au client qui a fait l'appel.

L'architecture Client-Serveur :



Les clients enfilent des requêtes dans la file synchronisée en respectant les contraintes suivant :

- L'accès à la file doit être protégé car elle s'agit d'une ressource partagée.
- Enfiler une requête dans une file pleine est une opération bloquante.

Le Démon Défile les requête à partir de la file. Il affecte chaque requête a un thread parmi un ensemble de threads déjà crée. Si aucun thread n'est libre le signal SIGUSR1 est envoyé au client.

Un thread exécute la commande et écrit le résultat dans le tube nommé du client qui a lancé la commande.

II. Réalisation du projet :

Pour implémenté cette architecture on a utilisé les structures et les programmes suivant :

1. Les Requêtes :

La structure d'une requéte est construit d'un tableau de caractère qui contiendra la commande provenant du client, et un autre tableau de caractère qui contiendra le nom du tube nommé unique du client.
Elle contient aussi le PID pour garder l'id de processus courant. Cette ID sera utilisé ultérieurement pour envoyer le signal SIGUSR1 s'il n'y a pas de ressource libre au niveau du serveur.

```c
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

typedef struct {

    //nom du tube du client
    char nom[256];
    //commande envoyÃ© par le client
    char cmd[256];
    //id du processus
    pid_t processId;
} request_t;

//mÃ©thode pour remplire une requette
void set_request(request_t *request,char *nom, char *cmd, pid_t i) {

    strcpy(request->nom, nom);
    strcpy(request->cmd, cmd);
    request->processId = i;

}
```

2. La file synchronisée :

Notre file peut être vue comme un tableau de requête, avec des indices **Head** et **Tail** pour gérer l'espace dans la file, elle a une taille fixe et propose deux opérations :

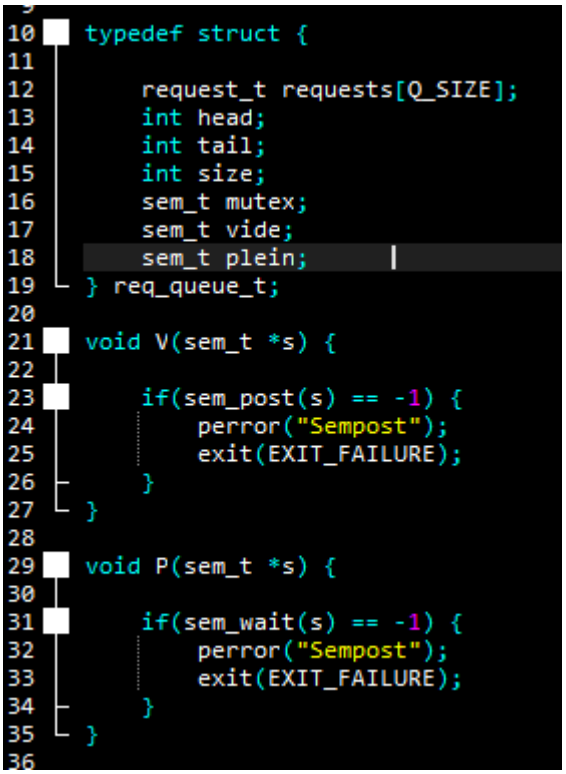
- Enfiler exécuter par un client.
- Défiler exécuter par le Démon (Serveur).

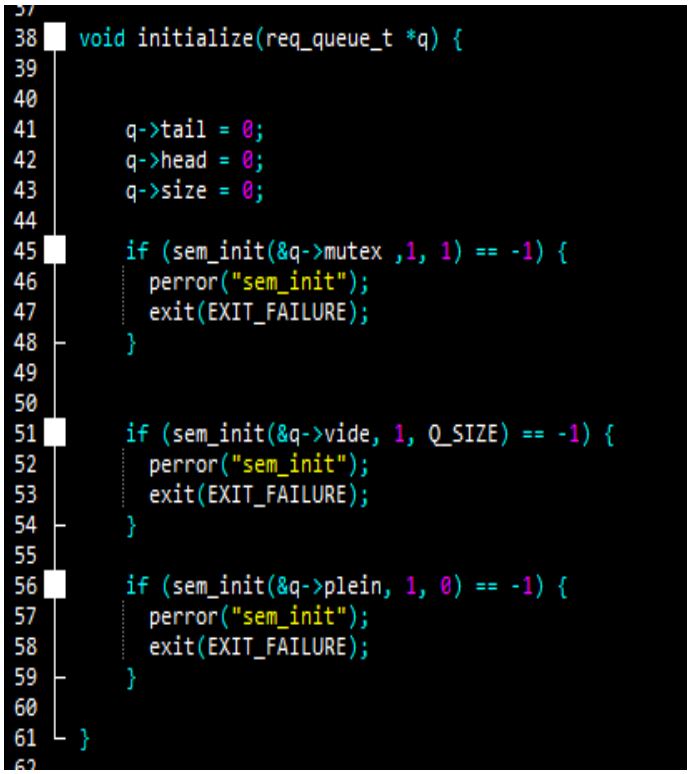
```c
int enfiler(req_queue_t *q, request_t r) {

    /*if(q->size == Q_SIZE)
      return Q_FULL;*/
    P(&q->vide);
    P(&q->mutex);

    q->requests[q->head] = r;
    q->head = (q->head + 1 ) % Q_SIZE;
    q->size++;

    V(&q->mutex);
    V(&q->plein);

    return SUCCESS;
}

request_t defiler(req_queue_t *q) {

     request_t r;

    P(&q->plein);
    P(&q->mutex);

    r = q->requests[q->tail];
    q->tail = (q->tail + 1) % Q_SIZE;
    q->size--;

    V(&q->mutex);
    V(&q->vide);


    return  r;
}
```

Ces deux opérations sont gérées par un mutex qui protège l'accès à la file.
En plus de ca on utilise des sémaphores plein vide pour gérer les deux cas suivant :

- Enfiler dans une file plein est une opération bloquante.
- Défiler dans une file vide est une opération bloquante.

On plus de ça nous avons ajoutés une méthode **initialize** pour initialiser la file et les sémaphores.





3. Segment de mémoire partagée :

Notre segment de mémoire partagée contient :

- La file synchronisée
- Une variable **Client_Index** qui sera utile lors de la création d'un nom unique du tube nommé de chaque client.
- Un sémaphore pour protéger l'accès à la variable **Client_Index** dans le programme Client.

```c
typedef struct {

    req_queue_t req_queue;
    volatile int client_index;
    sem_t *semIndex;
} shm_struct;


void VSHM(sem_t *s) {

    if(sem_post(s) == -1) {
        perror("SemPost");
        exit(EXIT_FAILURE);
    }

}

void PSHM(sem_t *s) {

    if(sem_wait(s) == -1) {
        perror("SemWait");
        exit(EXIT_FAILURE);
    }
}

```

4. Clients :

Notre programme client respect l'architecture suivante :

- Ouvrir le segment de mémoire partagé créé au lancement du serveur.

```c
    //ouverture de la mÃ©moire partagÃ©
    if((shm_fd = shm_open(SHM_NAME, O_RDWR ,0666) )== -1 ) {
        perror("shm_openssss");
        exit(EXIT_FAILURE);
    }
    //projection de la mÃ©moire partagÃ© sur l'espace d'adressage
    if((shm= (shm_struct *) mmap(NULL, sizeof(shm_struct), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0)) == MAP_FAILED) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
```

- Création du nom unique du tube nommé en concaténant le mot « Client » et l'indice client défini dans la structure du segment de la mémoire partagée.
  L'accès à la mémoire partagé pour récupère l'indice client est protégé par un sémaphore.
- Création du tube nommé.

```c
    //initialisation du semaphore sur l'indexe du client de shm
    shm->semIndex = malloc(sizeof(sem_t));
    if (sem_init(shm->semIndex, 1, 1) == -1) {
      perror("sem_init");
      exit(EXIT_FAILURE);
    }


    PSHM(shm->semIndex);

    //IncrÃ©mentation de l'indexe de clients
    shm->client_index++;
    //Construction du nom de tube
    snprintf(pipeName, 9, "%s%d",PIPE_NAME, shm->client_index);

    VSHM(shm->semIndex);


    //Creation du tube nommÃ©
    if (mkfifo(pipeName, S_IRUSR | S_IWUSR) == -1) {

        perror("mkfifo");
        exit(EXIT_FAILURE);
    }
```

- Dans une boucle infinie :
    - Construction de notre requête par la commande saisie par l'utilisateur, le nom du tube nommé et le PID du processus courant.
    - Enfiler la requête construit.
    - On traite si le serveur envoie le signal SIGUSR1 : si oui on affiche un message dans le client indiquant que les ressources du serveur sont occupées.

```
95      printf("\ncmd:");
96      char cmd[256];
97      //Get client input
98      fgets(cmd,256,stdin);
99      strtok(cmd, "\n");
100     //rÃ©cupÃ©rer l'id du processus courant
101     processId = getpid();
102     //remplire la requette Ã  enfiler
103     set_request(&requete, pipeName, cmd,processId);
104     //enfiler la requette
105     enfiler(requestQ, requete);
106     //manupulation de signal
107     signal(SIGUSR1,my_handler);
108     signal(SIGUSR2,my_handler2);
```

```
//fonction Ã  executer lors de reception d'un signal SIGUSR1 ou SIGUSR2
void my_handler(int signum) {

    if (signum == SIGUSR1) {
        write(1, "Pas de thread disponible\n", 32);
    } else if (signum == SIGUSR2){

        write(1, "serveur Ã©teint \n", 19);
        if(unlink(pipeName) == -1) {
            perror("unlink");
            exit(EXIT_FAILURE);
        }
        exit(0);
    }
}
```

- Ouverture du tube nommé en lecture
    - Dans une autre boucle : Lecture du résultat à partir du tube nommé, et l'écriture de cette dernière dans la sortie standard du client
    - Fermeture du tube nommé
- Suppression du nom associé au tube : **Unlink**.
- Fermeture du segment de la mémoire partagée.

```
//Ouverture de tube en lecture
out = open(requete.nom, O_RDONLY);
if(out == -1) {
    perror("open");
    exit(EXIT_FAILURE);
}

//lecture des donnÃ©es envoyÃ©es par le serveur
while (read(out, buffer, sizeof(buffer)) != 0) {

        write(1, buffer, strlen(buffer));
        memset (buffer, 0, sizeof(buffer));
    }

//Fermeture de tube
if(close(out) == -1) {

    perror("close");
    exit(EXIT_FAILURE);
}
```

```
//supression de tube nommÃ©
if(unlink(requete.nom) == -1) {
        perror("unlink");
        exit(EXIT_FAILURE);
    }
//fermeture de la mÃ©moire partagÃ©
if (close(shm_fd) == -1) {
    perror("close");
    exit(EXIT_FAILURE);
}

}
```

5. Le groupe des threads :

Threadgrp_t est une structure qui contient :

- Un tableau de thread de taille fixe pour simuler le fait qu'un serveur a un nombre limité de ressources qu'il faut gérer.
- Un tableau de booléen, de même taille qui indique si le thread est libre ou pas.
- Un tableau de requête, de même taille qui sera partagé entre le démon et l'ensemble des threads.
- Un tableau de **mutex** pour synchronisé les threads.

```
11    #define THREADS_NBR 30
12    typedef struct {
13
14        pthread_t threads[THREADS_NBR];
15        bool free[THREADS_NBR];
16        request_t data[THREADS_NBR];
17        sem_t mutex_tab[THREADS_NBR];
18        int indexes[THREADS_NBR];
19
20    } threadgrp_t;
```

En plus de la structure on implémente les méthodes suivantes :

- La méthode **CreateThreads** dans laquelle :
  - On initialise chaque élément dans le tableau de mutex par 0.
  - On crée les threads en envoyant l'indice avec lequel on parcourt les tableaux comme paramètre de la fonction run.
  - On initialise le tableau des threads par les threads qu'on vient de créer.
  - On initialise le tableau de booléen par **True** pour indiquer que tous les threads sont libres.

```
24  threadgrp_t *CreateThreads(threadgrp_t *tg) {
25      //creation des thread
26      int errnum;pthread_t th;
27
28      for(int j = 0; j<THREADS_NBR ; j++) {
29
30          tg->indexes[j] = j;
31          if (sem_init(&tg->mutex_tab[j], 1, 0) == -1) {
32              perror("sem_init");
33              exit(EXIT_FAILURE);
34          }
35
36          if((errnum = pthread_create(&th, NULL,run, &tg->indexes[j])) != 0) {
37
38              fprintf(stderr, "pthread_create: %s\n",strerror(errnum));
39          }
40
41          tg->threads[j] = th;
42          tg->free[tg->indexes[j]] = true;
43      }
44      return tg;
45  }
```

- La méthode **FreeIndex** qui parcourt le tableau de booléen et retourne l'indice du premier thread libre. Et il renvoie -1 si aucun thread n'est libre.
- Les méthodes PTH et VTH qui utiliserons le tableau de sémaphores pour synchroniser l'exécution de la méthode **run** des threads.

```c
int FreeThIndex(threadgrp_t *tg) {

    for(int i=0;i<THREADS_NBR;i++ ) {

        if(tg->free[i] == true)
        return i;
    }

    return -1;
}
```

```c
void PTH(sem_t *s){

    if(sem_wait(s) == -1) {

            perror("SemWait");
            exit(EXIT_FAILURE);
    }
}

void VTH(sem_t *s) {

    if(sem_post(s) == -1) {
            perror("SemPost");
            exit(EXIT_FAILURE);
    }
}
```

6. Démon :

Le démon ou le serveur est la partie du code où on va traiter les commandes entré par le client.
Pour cela :

- On crée un segment de mémoire partagé avec un nom bien défini et on alloue de l'espace mémoire pour notre structure **thread_grp**.
- On crée les threads.
- On initialise la file synchronisé.

```c
//allocation mÃ©moire pour le pointeur du groupe de thread
thread_grp = malloc(sizeof(threadgrp_t)*THREADS_NBR);

//MÃ©moire partagÃ©

if((shm_fd = shm_open(SHM_NAME, O_RDWR |O_CREAT,0666) )== -1 ) {

    perror("shm_open");
    exit(EXIT_FAILURE);
}
if (ftruncate(shm_fd, sizeof(shm_struct)*30) == -1) {

    perror("ftruncate");
    exit(EXIT_FAILURE);
}
if((shm = (shm_struct *) mmap(NULL, sizeof(shm_struct), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0)) == MAP_FAILED) {
    perror("mmap");
    exit(EXIT_FAILURE);
}

//creation des threads
thread_grp = CreateThreads(thread_grp) ;
//pointer sur la file de la mÃ©moire partagÃ©e
requestQ = &shm->req_queue;

//initialisation de la file
initialize(requestQ);
```

- Dans une boucle infinie :
  - ✓ on défile la première requête à partir de la file
  - ✓ On récupère l'indice du premier thread libre en faisant appel à la méthode **FreeThIndex**.
  - ✓ Dans le cas où il n'y a pas de ressources libre, on envoie au client le signal **SIGUSR1**.
  - ✓ Dans le cas contraire, on attribue la requête défilé au thread libre on utilisant le tableau de requête Data qui est partagé entre le démon et les threads.
  - ✓ On change l'état du thread pour marquer qu'il n'est plus libre.
  - ✓ Finalement on incrémente la valeur du **mutex** on appelant VTH, pour que le thread puisse exécuter sa méthode run.

```
//boucle principale du sÃ©rveur
while(1) {

    //rÃ©cupÃ©ration de la premiere commande arrivÃ©e
    req = defiler(requestQ);
    //rÃ©cupÃ©rer l'index d'un thread libre
    i = FreeThIndex(thread_grp);
    //envoyer un signal SIGUSR1 au client s'il n'y a pas de thread libre
    if(i==-1) {

        if (kill(req.processId,SIGUSR1) != 0) {

            printf("\nEnvoi de signal Ã©chouÃ©!\n");
        }
    }
    //attribution de la requÃªte rÃ©cupÃ©rÃ©e au thread libre
    thread_grp->data[i] = req;
    //changer l'Ã©tat du thread dans la structure thread_group
    thread_grp->free[i] = false;

    VTH(&thread_grp->mutex_tab[i]);
}
```

- Dans la méthode run des threads :
  - ✓ On fait appel à la méthode **parseCmd**, qui parse la partie commande de la requête pour extraire la commande et ces attributs.

```
143  // function for parsing command words
144  void parseCmd(char* str, char** parsed) {
145
146      int i;
147      for (i = 0; i < 256; i++) {
148
149          parsed[i] = strsep(&str, " ");
150          if (parsed[i] == NULL)
151              break;
152          if (strlen(parsed[i]) == 0)
153              i--;
154      }
155  }
```

```
void *run(void * arg) {

    int x = *(int *)arg;
    int *cmd_index = (int*)malloc(sizeof(int));


    while(true) {
        PTH(&thread_grp->mutex_tab[x]);


        char *parsed_cmd[256];
        request_t req = thread_grp->data[x];
        // execution en cas d'une commande sans pipe
        parseCmd(req.cmd, parsed_cmd);
```

  - ✓ On teste si la commande entré est particulière : **exit** (pour quitter un client), **turnoff** (pour arrêter le serveur).
  - ✓ On fait appel à la méthode **ExecuteCmd**, qui fait un fork et redirige la sortie standard vers le tube nommé dont le nom est passé avec la requête.

```c
// Function where the system command is executed
void ExecuteCmd(char** parsed, char *name) {

    // Forking a child
    pid_t pid = fork();
    if (pid == -1) {

        printf("\nFailed forking child..");
        return;
    } else if (pid == 0) {

        int in = open(name, O_RDWR, S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);
        if(in==-1) {

          perror("open User pipe");
          exit(EXIT_FAILURE);
        }

          dup2(in,STDOUT_FILENO);

        if(close(in) == -1) {
          perror("close");
          exit(EXIT_FAILURE);
        }


        if (execvp(parsed[0], parsed) < 0) {
          printf("\nCould not execute command..");
        }
        dup2(STDOUT_FILENO,STDOUT_FILENO);
        exit(0);


    } else {
        // waiting for child to terminate
        wait(NULL);
        return;
    }
}
```

```c
    if(simpleCMD(parsed_cmd,cmd_index)) {

      switch (*cmd_index) {

        case 0:
                kill(req.processId,SIGKILL);
                if(unlink(req.nom) == -1) {
                   perror("unlink");
                   exit(EXIT_FAILURE);
                }
                break;

        case 1:
                kill(req.processId,SIGUSR2);
                if (shm_unlink(SHM_NAME) == -1) {
                   perror("shm_unlink");
                   exit(EXIT_FAILURE);
                }
                printf("User %d turned off the server.\n",req.processId);
                exit(0);
                break;

        default:printf("whyyyyy");break;
      }

    }else {
      ExecuteCmd(parsed_cmd,req.nom);
    }

    thread_grp->free[x] = true;

    }

    pthread_exit(NULL);
}
```

```c
bool simpleCMD(char **parsed, int *cmd_index) {
    bool is_simple = false;
    char * simple_cmds[2] = {"exit","turnoff"};



    for (int c = 0; c < 2; c++) {
        if (strcmp(parsed[0], simple_cmds[c]) == 0) {

            *cmd_index = c ;
            is_simple = true;
            break;
        }
     }

return is_simple;
}
```

- ✓ On change l'état du thread pour marquer qu'il est à nouveau libre.
- ✓ Finalement, on termine l'exécution du thread courant.

```c
    thread_grp->free[x] = true;

    }

    pthread_exit(NULL);
}
```