PROJET SYSTÈMES D'EXPLOITATION

MAXIME DOURLEN
THÉO FINOT

Lors du semestre 5 de l'année scolaire 2021/2022 les étudiants en Licence 3 Informatique et Informatique parcours Science des Données devaient rendre un projet pour la matière Systèmes d'exploitation. Pour ce projet nous devions réaliser un démon d'information du système, c'est-à-dire un programme qui se charge de récupérer et traiter des requêtes. Un client peut émettre une requête pour demander les informations concernant un utilisateur à partir de son uid ou de son nom (login, nom réel, groupe, répertoire dédié, shell utilisé, . . .) , les informations sur un processus en fonction de son pid (commande, propriétaire, état, . . .), et toute commande système usuelle. Ce projet a été réalisé par Maxime Dourlen étudiant en Informatique TD3 TP5 et Théo Finot étudiant en Informatique parcours Science des Données TD4 TP7. Nous avons décider de faire ce projet ensemble car nous avions déjà réalisé des projets informatique lors de nos années précédentes ce qui nous a permis, lors de ce projet, d'être plus productif.
Comme lors des TP effectués ce semestre ce projet a était réaliser en langage C afin d'utiliser nos connaissances acquises lors de ces séances.

Nous avons commencé par créer un sémaphore, dans le fichier sema.c, afin de gérer la coordination entre les threads qui permettrons d'exécuter les requêtes émises par les clients. Voici un aperçu de la structure de données utilisée pour ce sémaphore appelée 'fifo' :

```
 2
 3  struct fifo {
 4      sem_t mutex;
 5      sem_t vide;
 6      sem_t plein;
 7      int tete;
 8      int queue;
 9      int buffer[];
10  };
11
```

Image de la structure de données 'fifo'

Il y a premièrement la variable 'mutex' de type sem_t qui contrôle l'accès au tampon ,que nous verrons ensuite, puis les variables de type sem_t 'vide' et 'plein' qui correspondent respectivement a compter les emplacements vides et occupées dans le tampon, la structure fifo comporte aussi les variables entières 'tete' et 'queue' qui sont la position d'ajout dans le tampon et la position de suppression dans le tampon. Et pour finir la structure fifo a aussi un tableau d'entier 'buffer' qui est le tampon contenant les données.

De plus dans ce fichier sema.c nous avons coder un algorithme producteur-consommateur afin de gérer restreindre l'accès a la zone critique et permet de donner accès a un thread. Dont voici le code :

```
102  void producteur(void) {
103    ssize_t n;
104    char c;
105    while ((n = read(STDIN_FILENO, &c, 1)) > 0) {
106      ajouter(c);
107    }
108    if (n == -1) {
109      perror("read");
110      exit(EXIT_FAILURE);
111    }
112    ajouter(0);
113  }
114
115  void consommateur(void) {
116    pid_t pid;
117    do {
118      pid = retirer();
119      errno = 0;
120      if (write(STDOUT_FILENO, &pid, 10) < 1) {
121        if (errno != 0) {
122          perror("write");
123          exit(EXIT_FAILURE);
124        } else {
125          fprintf(stderr, "write: impossible d'écrire les données demandées");
126          exit(EXIT_FAILURE);
127        }
128      }
129    } while (pid != 0);
130  }
131
```

image de l'implémentation de l'algorithme producteur-consommateur

Ensuite dans le fichier client.c nous gérons les requêtes émises par les clients que nous écrivons ensuite dans un tube que le démon lira afin de l'exécuter et ensuite nous lisons la réponse envoyer par le démon via un tube. Pour cela dans la fonction main du code contenu dans le fichier client.c nous créons deux tubes nommés, le premier pour envoyer la requêtes au démon et le second afin de récupérer la réponse de la requêtes et ensuite nous supprimons ses tubes. Nous avons coder les fonction 'namegenrator' et 'tube_delete' afin de générer un nom aux tubes afin de les les nommés et ainsi mieux les gérer, et 'tube_delete' permet de supprimer les tubes à la fin de leur utilisation.

```
100  int namegenerator(pid_t pid, char *result, int n) {
101    char mypid[9];
102    if (sprintf(mypid, "%d", pid) < 0) {
103      return EXIT_FAILURE;
104    }
105    if (n == 0) {
106      strcpy(result, TUBE_RQT);
107    } else {
108      strcpy(result, TUBE_RPS);
109    }
110    strcat(result, mypid);
111    printf("%s\n", result);
112    return EXIT_SUCCESS;
113  }
114
```

Image de la fonction 'namegenrator'

```c
int tube_delete() {
  pid_t pid = getpid();
  char buffer1[MAX_TN];
  char buffer2[MAX_TN];
  if (namegenerator(pid, buffer1, 0) != 0) {
    exit(EXIT_FAILURE);
  }
  if (namegenerator(pid, buffer2, 1) != 0) {
    exit(EXIT_FAILURE);
  }
  if (unlink(buffer1) == -1) {
    perror("unlink");
    exit(EXIT_FAILURE);
  }
  if (unlink(buffer2) == -1) {
    perror("unlink");
    exit(EXIT_FAILURE);
  }
  return EXIT_SUCCESS;
}

```

Image de la fonction ‘tube_delete’

Voici une image illustrant l’utilisation de namegenrator et et ensuite de la création du tube ouvert en écriture afin d’écrire la requête pour le démon :

```c
  char *buf1 = malloc(MAX_TN);
  if (namegenerator(getpid(), buf1, 0) != EXIT_SUCCESS) {
    exit(EXIT_FAILURE);
  }
  if (mkfifo(buf1, S_IRUSR | S_IWUSR) == -1) {
    perror("mkfifo");
    exit(EXIT_FAILURE);
  }
```

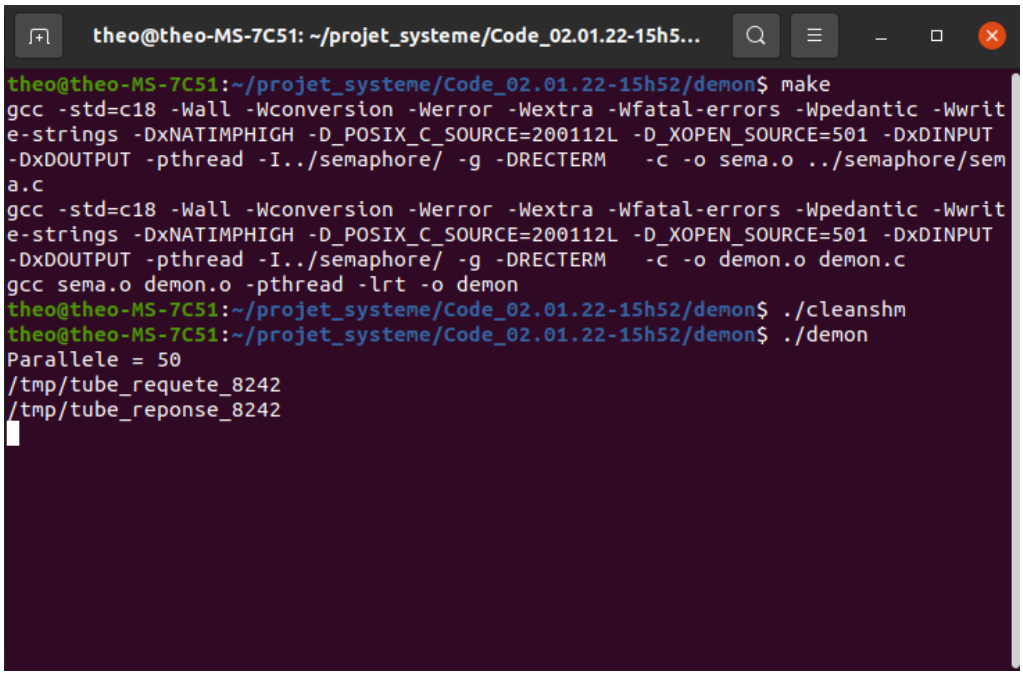
Image de l’utilisation de namegenrator et de l’ouverture du tube en écriture

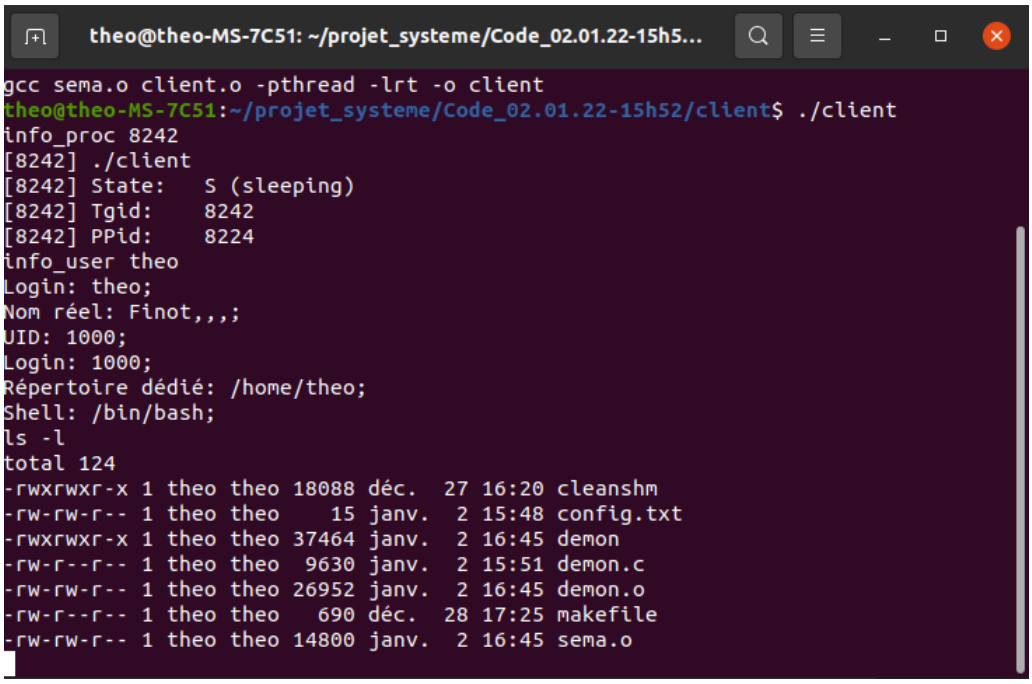
Et enfin nous avons implémenter le code du démon dans le fichier ‘demon.c’ qui lis la requêtes d’un client dans un tube, ensuite exécute cette requêtes dan un ‘fork’, et écris le résultat de l’exécution dans un tube qui est envoyé au ‘client.c’.

```c
  switch (fork()) {
    case -1:
      perror("fork");
      semaphore_serv_close();
      exit(EXIT_FAILURE);
    case 0:
      if (close(fd1) == -1) {
        semaphore_serv_close();
        perror("close");
        exit(EXIT_FAILURE);
      }
      if (dup2(fd2, STDOUT_FILENO) == -1) {
        semaphore_serv_close();
        perror("dup2");
        exit(EXIT_FAILURE);
      }
      if (dup2(fd2, STDERR_FILENO) == -1) {
        semaphore_serv_close();
        perror("dup2");
        exit(EXIT_FAILURE);
      }
      execlp("/bin/sh", "/bin/sh", "-c", rbuf2, (char *) NULL);
      perror("exec");
      semaphore_serv_close();
      exit(EXIT_FAILURE);
    default:
      break;
  }
```

On peut voir sur cette image que la fonction ‘execlp’ ligne 140 exécute la requête du client qui est stocké dans la variable ‘rbuf2’.

Exemple de l’exécution du projet :





Durant la réalisation de ce projet nous n’avons pas rencontrer de problème majeur, malgré quelques difficultés lors de la gestion des signaux et des threads et aussi des difficultés afin d’éviter les inter blocages lors de l’ouverture des tubes. Pour conclure, notre projet répond aux demandes du sujet et satisfait les contraintes de la compilation, de Valgrind, et toutes autre contraintes du sujet.