Rapport SR02 : DEVOIR 1

Macéo ANGER
Léopold CHAPPUIS

Sujet : Écrivez un programme en langage C sous Unix qui simule un système de gestion de tâches basique. Le programme doit inclure les fonctionnalités suivantes :

1 : Le processus principal crée un processus fils, appelé "gestionnaire de tâches".

On utilise classiquement un fork() dans le main. Ainsi le processus du main (le père) crée un processus fils qui sera le gestionnaire de tâche. On initialisera également une variable pid pour sauvegarder le pid du gestionnaire afin d'intervenir dessus via le père.

```c
pid_t pid;
int main() {
    pid = fork();
    if (pid == 0) {
        //fils (gestionaire de tâche)
    } else if (pid > 0) {
        //PERE
    } else {
        perror("fork");
        exit(1);
    }
    return 0;
}
```

2 : Le gestionnaire de tâches créer une file d'attente (par exemple, une pipe) pour recevoir des demandes de tâches provenant d'autres processus.

On utilise un pipe qu'on initialise dans le main int fd[2]. Lorsque le père reçoit une tâche, il écrit dans la pipe (tête d'écriture) via un write(). Le gestionnaire de tâche va lui lire la tête de lecture de la pipe via un read() pour recevoir la tâche à exécuter. On implémente la structure et l'exécution de "tâche" dans les questions suivantes.

```c
int main() {
    int fd[2];
    //Créer une pipe pour la communication entre le
    //processus principal et le gestionnaire de tâches
    if (pipe(fd) == -1) {
        perror("pipe");
        exit(1);
    }
    pid = fork();
    if (pid == 0) { //  fils (gestionaire de tâche)
        close(fd[1]);
        while(1){
            char var[50];
            //Recupere la tache
            read(fd[0], var, sizeof(var));
            sleep(2);
        }
        exit(0);
```

```c
    else if (pid > 0) {
        close(fd[0]);
        while (1) {
            if("tache" != NULL){
                write(fd[1], "tache N°X", 50);
            }else{
                write(fd[1], "pas de tache", 50);
            }
            sleep(1);
        }
    } else {
        perror("fork");
        exit(1);
    }
    return 0;
}
```

3 : Le processus principal doit également installer un gestionnaire de signal pour capturer le signal SIGINT (Ctrl+C).
4 : Lorsque le signal SIGINT est reçu par le processus principal, celui-ci doit envoyer un signal SIGUSR1 au gestionnaire de tâches pour lui indiquer de terminer son exécution.

On utilisera pour cette question les structures sigactions. Dans un premier temps, on initialise un gestionnaire de signal pour le fils (gestionnaire de tâche). On paramètre le gestionnaire pour qu'il attende le signal SIGUSR1 puis on définit une fonction qui termine le processus en cas de réception du SIGUSR1. Ensuite on initialise un gestionnaire de signal pour le père. On paramètre ce gestionnaire sur le signal SIGINT (le ctrc+c). Enfin, on définit une fonction d'attente de ce signal (SIGINT) et qui envoie le signal SIGUSR1 au fils (gestionnaire de tâche). Ainsi lorsque le père reçoit le signal SIGINT, il envoie le signal SIGUSR1 au fils qui arrête son exécution.

```c
struct sigaction pere;
struct sigaction fils;
fils.sa_sigaction = handle_sigusr1;
    fils.sa_flags = SA_SIGINFO;
    sigemptyset(&fils.sa_mask);
    if (sigaction(SIGUSR1, &fils, NULL) == -1) {
        perror("sigaction fils");
        exit(1);
    }
pere.sa_sigaction = handle_sigint;
    pere.sa_flags = SA_SIGINFO;
    sigemptyset(&pere.sa_mask);
    if (sigaction(SIGINT, &pere, NULL) == -1) {
        perror("sigaction pere");
        exit(1);
    }
```

```c
int verif = 1;
void handle_sigint(int sig, siginfo_t *info, void *context) {
    if(!verif){
        printf("\nLe processus gestionnaire de tâche a déjà été terminé !\n");
    }
    if(pid != 0){
        kill(pid, SIGUSR1);
        verif = 0;
    }
}

void handle_sigusr1(int sig, siginfo_t *info, void *context) {
    printf("Processus gestionnaire de tâche terminé.\npid : %d \n", getpid());
    exit(0);
}
```

6 : Chaque demande de tâche doit être encapsulée dans une structure de données appropriée, contenant au moins l'identifiant de la tâche et ses paramètres.

On utilisera une structure en chaînage avant pour stocker les différentes tâches. Chaque tâche est une structure de données comportant un identifiant de tâche (int), des paramètres (String), le chaînage suivant (Pointeur). On définit alors des fonctions qui crée le premier élément de chainage, qui ajoute un élément à la chaîne, et qui supprime un élément de la chaîne.

```c
typedef struct {
    int task_id;
    char* params;
    struct TaskRequest* next;
} TaskRequest;
```

```c
TaskRequest* deleteFirstTask(TaskRequest* file){
    TaskRequest* tmp = file->next;
    free(file);
    return tmp;
}
```

```c
TaskRequest* createTask(char* params, unsigned int id){
    TaskRequest* newTask = malloc(sizeof(TaskRequest));
    if (newTask == NULL) {
        return NULL;
    }
    newTask->next = NULL;
    newTask->task_id = id;
    newTask->params = params;
    return newTask;
}
```

```c
void addTask(TaskRequest *file, char* params, unsigned int id){
    TaskRequest* tmp = file;
    while(tmp->next != NULL){
        tmp = tmp->next;
    }
    tmp->next = createTask(params, id);
}
```

5 : Les autres processus (appelés "clients de tâches") peuvent envoyer des demandes de tâches au gestionnaire de tâches via la file d'attente.

Le père génère ou récupère des tâches selon l'implémentation, pour l'exemple, on va générer 5 tâches. Les tâches sont alors traitées/placées dans la file d'attente 1 par 1 via le chaînage.

```
// père
close(fd[0]);
TaskRequest* file = createTask("Task 1", 1);
addTask(file, "Task 2", 2);
addTask(file, "Task 3", 3);
addTask(file, "Task 4", 4);
addTask(file, "Task 5", 5);
while (1) {
    printf("Le père vit ! \n");
        if(verif){
            if(file != NULL){
                write(fd[1], file->params, 50);
                file = deleteFirstTask(file);
            }
            else{
                write(fd[1], "pas de tache", 50);
            }
        }else{
            // le gestionnaire de tâche a été tué, ne rien faire
        }
    sleep(1); // Simule une pause entre les envois de demandes
}
```

7 : Le gestionnaire de tâches doit traiter les demandes de tâches dans l'ordre d'arrivée et exécuter chaque tâche dans un processus enfant séparé (utilisation de fork).

Le fils (gestionnaire de tâches) attend que la tête de lecture de la pipe soit différente de "pas de tache". Lorsqu'il y'a des tâches à exécuter, le gestionnaire crée via un fork() des sous fils qui exécute une tâche (pour l'exemple, on affiche un message via printf()). Le gestionnaire de tâche est limité en nombre de processus à exécuter simultanément. Ainsi on compte chaque sous fils pour tester cette contrainte et le gestionnaire va se mettre en attente si le maximum est dépassé.

```
//  fils (gestionaire de tâche)
close(fd[1]);
int active_children = 0;
while(1){
    char var[50];
    read(fd[0], var, sizeof(var));
    if(strcmp(var,"pas de tache") == 0 || strcmp(var, "") ==0){
        printf("en attente de tache \n");
    }else{
        pid_t child_pid = fork();
        if (child_pid == 0){
            // Fils
            printf("Execution de la tâche : %s \nPID du processus executant la tache : %d \n",var, getpid());
            exit(0);
        } else if (child_pid > 0) {
            // Père
            active_children++;
            if (active_children >= MAX_CHILD_PROCESSES) {
                // Attendre qu'un processus fils se termine avant de continuer
                wait(NULL);
                active_children--;
            }
        } else {
            perror("fork");
            exit(1);
        }
    }
    sleep(2);
}
exit(0);
```

8 : Une fois qu'une tâche est terminée, le gestionnaire de tâches doit envoyer un message de retour au client de tâches appropriées pour signaler l'achèvement de la tâche.

Dans le contexte de l'exercice, nous avons utilisé la fonction exit(0) (sans erreur) pour un retour au client de tâche.

9 : Le gestionnaire de tâches doit également gérer les cas où un processus enfant dépasse un certain temps d'exécution et le tuer si nécessaire, en envoyant un signal approprié au processus enfant.

Nous avons défini un timer (TIME_OUT_PROCESS) de X seconde dans un #définе, à chaque création d'un sous fils, le gestionnaire de tâches envoie une alarme au bout de ce X secondes. Grâce à un gestionnaire de signal, le sous fils toujours en exécution après ce laps de temps est tué.

```
fils_alarm.sa_sigaction = handle_alarm;
fils_alarm.sa_flags = SA_SIGINFO;
sigemptyset(&fils_alarm.sa_mask);
if (sigaction(SIGALRM, &fils_alarm, NULL) == -1) {
    perror("sigaction fils alarm");
    exit(1);
}

alarm(TIME_OUT_PROCESS);
```

```
void handle_alarm(int sig, siginfo_t *info, void *context) {
    if (kill(child_pid, 0) == 0) {
        printf("\tProcessus %d time out\n", child_pid);
        kill(child_pid, SIGTERM);
    }
}
```

Pour tester notre gestionnaire, nous avons décidé de créer une boucle qui demande à l'utilisateur des commandes linux à exécuter. Et les commandes sont exécutées via la fonction system() une par une.

```
while(1) {
    memset(temp, 0, sizeof(char)*50);
    memset(task, 0, sizeof(char)*50);

    printf("Rentrez une commande : ");
    fgets(temp, sizeof(temp), stdin);
    temp[strcspn(temp, "\n")] = 0;
    strcpy(task, temp);

    if(strcmp(task, "sendtasks") != 0) {
        addTask(file, task, compteur);
        compteur++;
        printf("Commande choisie : %s\n", task);
    } else {
        break;
    }
```