Devoir 2 SR01 : Programmation Système

A remettre le : 30 décembre 2022

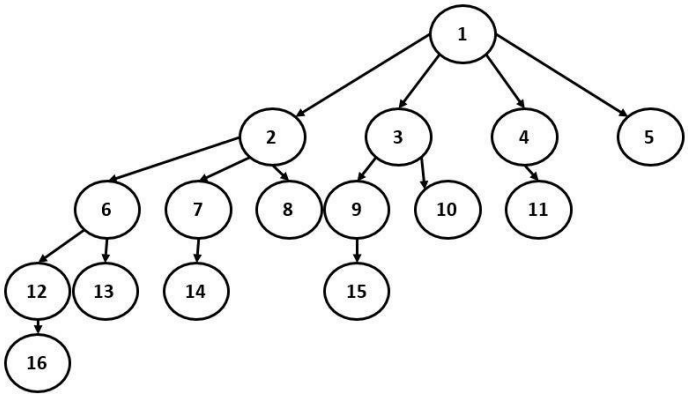
- Ce devoir doit se faire en binôme. Les deux étudiants formant un binôme doivent appartenir au même groupe de TD.
- Les différents binômes devront remettre un seul fichier NomPrenom1-NomPrenom2.zip contenant les codes sources ainsi qu'un rapport électronique (pdf) de quelques pages. Il est demandé de déposer un seul livrable par binôme pour faciliter les corrections
- Les noms des fichiers et les prototypes des fonctions doivent être respectés : nous utilisons des scripts pour lancer et tester vos codes.
- Les erreurs doivent être gérées (**<errno.h>, perror(), return des fonction, macros des fonctions**)
- Le dernier délai pour la remise des TP est fixé pour le 30 décembre 2022.

Exercice 1

Objectifs :

- Comprendre le fonctionnement de fork()
- Obtenir des informations

Soit l'arbre généalogique de processus suivant :



Question : En utilisant la fonction *fork()*, proposer un programme C **"Prog1.c"** permettant de générer cet arbre de processus. Pour chaque processus, afficher son PID et celui de son père sur une nouvelle ligne ***("Mon pid est : %d et le pid de mon père est : %d \n"***). On ne demande pas de synchroniser l'exécution des processus mais seulement obtenir exactement le même graphe hiérarchique des processus de l'image ci-dessus.

Exercice 2

Objectifs :

- Installer et manipuler Git
- Modularité et makefile
- Maitriser les appels système (fork, wait, etc.) et obtenir des informations
- Utiliser des commandes shell

Soit le programme en C suivant :

```c
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<string.h>
int premier(int nb){
    int r=0;
    for(int i = 1 ; i <= nb ; i++ ){
        if(nb % i == 0) r++;
    }
    if(r>2)
       return 0;
    else
       return 1;
}
void explorer(int debut, int fin){
  int etat,pid,pid2;
  pid=fork();
  if(pid==0){
    for (int i=debut; i<=fin;i++){
      if (premier(i)==1) {
        pid2=fork();
        if (pid2==0){
          char chaine[100];
          sprintf(chaine,"echo ' %d  est un nombre premier \
          écrit par le processus %d'>>nbr_premiers.txt",i,getpid());
          system(chaine);
          sleep(2);
          exit(0);
        }
        else wait(&etat);// instruction 41
      }

    }
    exit(0);
  } else wait(&etat);// instruction 46
}
int main(){
  int grp=1;
  while(grp<=11){
    explorer(grp+1,grp+10);
    grp=grp+10;
  }
}
```

Prog_premiers.c

On suppose dans ce programme que :

- le PID du shell est 100,
- le PID du processus correspondant au parent est 120,
- la numérotation des processus est séquentielle (incrémentation par 1).

Partie 1 : fonctionnement du programme ci-dessus :

1. Donner l'arbre généalogique des processus générés par ce programme (fork, system).
2. Donner une explication détaillée sur le fonctionnement du programme **Prog_premiers**.
3. Qu'est-ce que ça change si on supprime les instructions **41** et **46** ?
4. Comment ce programme arrive à garantir la synchronisation suivante : un processus de PID p ne peut être exécuté (***est exécuté =Il a fini d'exécuter system("chaine")***) qu'après la fin de l'exécution du processus de PID p - 1.

Partie 2 : implémentation du programme ci-dessous :

1. Installer git (interface terminal) sur votre machine Linux et documenter cette étape dans le rapport (captures d'écran, démarche, …).
2. Donner les instructions à exécuter pour la création d'une copie locale de dépôt existant suivant : ***https://gitlab.utc.fr/lounisah/devoir-2-a-22-sr-01***. Dans ce dépôt, il y a les ressources nécessaires pour cet exercice (Prog_premiers .c, Makefile, …).
3. Lancer le programme. Expliquer le contenu du fichier nbr_premiers.txt et les instructions/commandes qui ont permis d'avoir ce résultat (sprintf, system, echo, >>, …).

Partie 3 : modification, modularité et Makefile:

1. Est-ce que les pids des processus qui ont écrit dans le fichier nbr_premiers.txt sont correctement récupérés ? Sinon, modifier le code (***nommez-le Prog_premiers_m1.c***) qui édite le fichier nbr_premiers.txt pour (sur chaque ligne) :
    a. Écrire le pid du processus qui a créé le processus system (processus qui fait appel à la fonction system) et le père de ce processus (***processus qui exécute la boucle for***)
    b. Écrire le pid du processus (processus crée dans la fonction system, il est de type shell) qui a écrit le message sur le terminal.

2. Donner une autre version de ce programme (***nommez-le Prog_premiers_m2.c***) en utilisant excev() pour implémenter une fonction my_system(char *) qui remplace system().
3. Proposer une autre version (***nommez-le Prog_premiers_m3.c***) qui permet l'exécution des processus fils en parallèle tout en évitant les processus zombies.
4. Découper votre programme en plusieurs modules en respectant le Makefile suivant :

```
Prog_premiers : main.o explorer.o premier.o my_system.o
        gcc main.o explorer.o premier.o my_system .o –o
Prog_premiers
main.o : main.c explorer.h
        gcc –c main.c
explorer.o : explorer.c my_system.h explorer.h premier.h
        gcc –c explorer.c
premier.o : premier.c premier.h
        gcc –c premier.c
my_system.o : my_system.c my_system.h
        gcc –c my_system.c
clean :
        rm my_system.o premier.o explorer.o main.o Prog_premiers
```

Makefile

5. Lancer le Makefile avec la commande ***make*** et effectuer des tests.
6. Proposer la version la plus optimale « **MakefileOpt** » de Makefile précédent. Donner la commande qui permet de tester s'il fonctionne (modifier le contenu des printfs pour provoquer des mises à jour).
7. Donner les commandes git pour ajouter et valider sur votre dépôt local les modifications des questions : 1, 2, 3, 4, 6.

Exercice 3

Objectifs :

- Accès à un fichier via des fonctions C
- Création de plusieurs processus selon le besoin.
- Gestions des signaux et handlers

Un gestionnaire d'applications est un programme lancé au démarrage du système d'exploitation. Il se charge de lancer et de gérer un ensemble d'applications nécessaires au bon fonctionnement du système (gestion des cartes réseau, gestion des périphériques...). Dans cet exercice, on va programmer un gestionnaire d'applications personnalisé (*ApplicationManager*). La liste des applications à lancer est stockée dans le fichier

list_appli.txt. Vous disposez de quelques exemples d'applications (*power_manager.c*, *network_manager.c*, *get_time.c*).

Exemple de fichier *list_appli.txt* contenant deux applications :

```
nombre_applications=2
name=Power Manager
path=./power_manager
nombre_arguments=2
arguments=
./mise_en_veille.txt
4

name=Get
Time path =./get_time
nombre_arguments=0
arguments=
```

1. Écrire un programme *ApplicationManager.c* qui doit :
    - créer un ensemble de processus fils chacun est responsable à l'exécution d'une application,
    - lors de l'arrêt d'une application, informer l'utilisateur en lui affichant le nom de l'application terminée,
    - s'arrêter après avoir fermé toutes les applications en cours d'exécution lors de la réception d'un ordre de mise en veille de la part de *power_manager* (signal SIGUSR1).
2. Modifier le programme *power_manager.c* pour envoyer le signal SIGUSR1 à l'*ApplicationManger* lorsque l'utilisateur tape 1 dans le fichier *mise_en_veille.txt*

NB : Lorsque *ApplicationManger* reçoit un signal SIGUSR1 de la part d'un autre processus, il ne ferme pas les applications.