L3

Contrôle d'Architecture Système Avancé – 12 novembre 2013

Michel SOTO

AUCUN DOCUMENT AUTORISÉ

Durée: 1 H 30

Le barème est indicatif - Nombre de pages: 2 sur une feuille

La concision de vos réponses et la propreté de votre copie seront prises en compte.

PARTIE I : CONNAISSANCE DU COURS

Question 1 (4 points)

Répondez aux affirmations suivantes uniquement par "VRAI", ou "FAUX" ou "NE SAIS PAS".

Barème : **réponse exacte : +1 point, réponse fausse : -0,5 point sur la copie, "ne sais pas" : 0 point**

a) Un processus zombie devient orphelin quand sont père se termine. FAUX

b) Le système gère une table u_ofile par processus. VRAI

c) Par défaut, les fichiers ouverts sont fermés après l'appel à une primitive exec. FAUX

d) Un `fork` provoque la copie des données du processus père. FAUX (c'est la modification des données soit par le père soit par le fils qui provoque la copie des données : *copy on write*)

Question 2 (2 points)

Donnez le résultat de l'exécution du programme suivant. Justifiez votre réponse.

```
main(){
  int i;
  for (i=1;i<=2;i++){
      execl("ls","ls","-l",NULL);
  }
}
```

Il ne se passe rien car `execl` n'utilise pas la variable `PATH` et donc ne trouve pas l'exécutable de `ls` qui se trouve dans `/bin`

PARTIE II : APPLICATION DU COURS

Question 3 (4 points)

Soient deux fichiers appartenant à `Francois` du groupe `users` tels que décrits ci-dessous. `a.out` est un programme qui ouvre en lecture/écriture (`O_RDWR`) le fichier `Donnees`.

```
-rwsrwxr-x    1 francois  users     11687 déc  2 14:12 a.out
----rw-r--    1 francois  users        44 déc  2 14:09 Donnees
```

François et Pierre sont du même groupe. Le programme `a.out` peut-il ouvrir le fichier `Donnees` s'il est exécuté par :

a) 1) François ?

1. ID user effectif du processus est **DIFFERENT** de 0 (superuser),
2. ID user effectif (**Francois**) du processus est **EGAL** à ID propriétaire du fichier (**Francois**) :
   b. le mode d'accès (**O_RDWR**) **NE CORRESPOND PAS** aux droits d'accès (**---**) propriétaire (**Francois**), l'accès est **REFUSE**

b) 2) Pierre ?

1. ID user effectif du processus est **DIFFERENT** de 0 (superuser),
2. ID user effectif (**François car le set_user_id bit est positionné (rws) sur a.out**) du processus est **EGAL** de ID propriétaire du fichier (**Francois**)
   b. le mode d'accès (**O_RDWR**) **NE CORRESPOND PAS** aux droits d'accès (**---**) propriétaire (**Francois**), l'accès est **REFUSE**

c) On change le propriétaire du fichier Donnees ainsi :

```
----rw-r--    1 Pierre  users       44 déc  2 14:09 Donnees
```

Même question qu'en a)

1) François ?

1. ID user effectif du processus est **DIFFERENT** de 0 (superuser),
2. ID user effectif (**Francois**) du processus est **DIFFERENT** de ID propriétaire du fichier (**Pierre**)
3. ID groupe effectif du processus (**users**) du processus est **EGAL** à ID groupe du fichier (**users**) :
   a. le mode d'accès (**O_RDWR**) correspond aux droits d'accès du groupe (**rw-**), l'accès est **AUTORISE**

2) Pierre ?

1. ID user effectif du processus est **DIFFERENT** de 0 (superuser),
2. ID user effectif (**François car le set_user_id bit est positionné (rws) sur a.out**) du processus est **DIFFERENT** de ID propriétaire du fichier (**Pierre**)
3. ID groupe effectif du processus (**users**) du processus est **EGAL** à ID groupe du fichier (**users**) :
   b. le mode d'accès (**O_RDWR**) correspond aux droits d'accès du groupe (**rw-**), l'accès est **AUTORISE**

Dans les questions suivantes, les programmes devront être rédigés selon les règles de l'art : vérification du nombre de paramètres éventuels, vérification des valeurs de retour des primitives système et indentation du code. Vous êtes dispensé des `includes`.

Question 4 (4 points)

Vous devez écrire un programme qui, lors de son exécution, exécute N fois la commande **ps** puis affiche le message : `FIN DES N EXECUTIONS DE PS`

```
#Ie <stdio.h>
#Ie <stdlib.h>
#Ie <unistd.h>

main (int argc, char *argv[]){
int I, n, pid, status;

   if (argc < 2){
        // Utiliser perror ici n'a pas de sens !!
        fprintf(stderr, "Usage: %s N\n", argv[0]);
        exit(EXIT_FAILURE);
   }
   n=atoi(argv[1]);
   for (i=0; i<n; i++){
        pid=fork(); // Indispensable car sinon le 1er exec écraserait la boucle
        if (pid < 0){
             perror("kill");
             exit(EXIT_FAILURE);
        }
        if (pid==0){
             execlp("ps","ps",NULL); // execlP nécessaire pour utiliser le path
             perror("exec");
             exit(EXIT_FAILURE);
        }
        else {// Code du père
             pid=wait(&status);// Sans le wait, le message de fin peut
                               // apparaître avant la fin réelle de N exécutions
             if (pid < 0){
                  perror("wait");
                  exit(EXIT_FAILURE);
             }
        }
   }// for

   // Code du père
   printf("FIN DES %d EXECUTIONS DE PS\n", n);

} // main
```

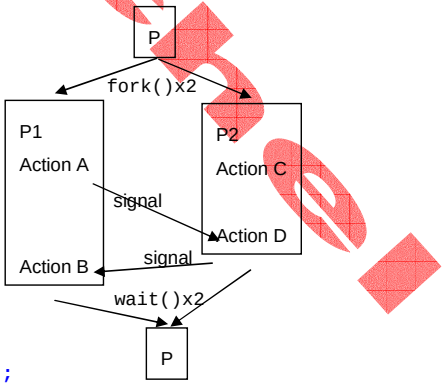
Question 5 (6 points)

Soit 2 processus P_1 et P_2 créés par même processus P. P_1 doit exécuter les actions A et B. P_2 doit exécuter les actions C et D. Les actions A et C doivent être exécutées en premier. Elles peuvent être exécutées en parallèle ou

dans n'importe quel ordre entre elles. L'action D ne doit être exécutée qu'après les actions A et C. Enfin l'action B ne doit être exécutée qu'après l'action D. Lorsque ses fils sont terminés, P affiche **FIN**.

En utilisant les signaux, écrivez en C le programme réalisant les actions A, B, C et D dans l'ordre voulu. Les actions elles-mêmes se limiteront à un **printf(P_i : action X\n)** aux endroits concernés du programme.

L'échange de signaux entre P1 et P2 est bidirectionnel. Etant donné que P1 et P2 sont frères, celui qui sera crée en premier ignorera l'identité (PID) du second (cf. TD). C'est donc P qui doit relayer le signal du premier vers le second.




```
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

#define VAS_Y_P1 SIGUSR1
#define VAS_Y_P2 SIGUSR2

int
      ok_b=0, ok_d=0, P1, P2, P;

/*---------------------*/
  void vas_y_P1()
/*---------------------*/
{// P2 informe P1 que D est terminée
 int k;
 printf("\t\t\t\tP2 %d: vas_y_P1\n", getpid());

 k=kill (P1, VAS_Y_P1);
 if (k<0)
     if (errno!=ESRCH) {// si errno==ESRCH c'est que P1 est terminé
          perror("P1 terminé");
          _exit (EXIT_FAILURE);
     }
}// vas_y_P1

/*---------------------*/
  void vas_y_P2 ()
/*---------------------*/
{// P1 informe P que A est terminée ou
 // P informe P2 que A est terminée
 int k;
 if (getpid()==P)
      printf("P %d: vas_y_P2\n", getpid());
 else printf("\t\tP1 %d: vas_y_P2\n", getpid());

 k=kill (P2, VAS_Y_P2);
 if (k<0)
     if (errno!=ESRCH) {// si errno==ESRCH c'est que P2 est terminé
          perror("P2 terminé");
          _exit (EXIT_FAILURE);
     }
}// vas_y_P2


/*---------------------*/
  void OK_D()
/*---------------------*/
{// P2 a recu le signal VAS_Y_P2
 // car l'action A de P1 est terminée
```

```
 // L'action D peut etre exécutée

    ok_d=1; // mémorisation que D peut être exécutée
}// OK_D

/*--------------------*/
  void OK_B()
/*--------------------*/
{// P1 a recu le signal VAS_Y_P1
 // car l'action D de P2 est terminée
 // L'action B peut etre exécutée

    ok_b=1; // mémorisation que B peut être exécutée
}// OK_B


/*--------------------*/
  void action_A() {// P1
/*--------------------*/
 printf("\t\tP1 %d: Execution action A\n",getpid());fflush(stdout);
}// action_A

/*--------------------*/
  void action_B() {// P1
/*--------------------*/
 printf("\t\tP1 %d: Execution action B\n", getpid());fflush(stdout);
}// action_B

/*--------------------*/
  void action_C() {// P2
/*--------------------*/
 printf("\t\t\t\tP2 %d: Execution action C\n", getpid());fflush(stdout);
}// action_C

/*--------------------*/
  void action_D() {// P2
/*--------------------*/
 printf("\t\t\t\tP2 %d: Execution action D\n", getpid());fflush(stdout);
}// action_D

/*--------------------*/
  void synchro_avec_P1 ()
/*--------------------*/
{// Fonction appelée par P et P2

 if (ok_d==0) { // A de P1 est-elle terminée ??
                // Ce test est INDISPENSABLE
                // car A peut déjà être terminée
                // quand on arrive ici
      if (getpid()==P2)
          {printf("\t\t\t\tP2 %d: J'attends\n", getpid());fflush(stdout);}

     else   {printf("P %d: J'attends\n", getpid());fflush(stdout);}
     pause(); // Attente que P1 ait terminée A
 }
 else if (getpid()==P2)
           {printf("\t\t\t\tP2 %d: Je n'attends pas\n", getpid());fflush(stdout);}
     else   {printf("P %d: Je n'attends pas\n", getpid());fflush(stdout);}

}// synchro_avec_P1

/*--------------------*/
  void synchro_avec_P2 ()
/*--------------------*/
{// Fonction appelée par P1
```



```
 if (ok_b==0) { // D de P2 est-elle terminée ??
                // Ce test est INDISPENSABLE
                // car A peut déjà être terminée
                // quand on arrive ici
     printf("\t\tP1 %d: J'attends\n", getpid());

     pause(); // Attente que P2 ait terminée D
 }
 else {printf("\t\tP1 %d: Je n'attends pas\n");fflush(stdout);}

}// synchro_avec_P2

/*=========================================================*/
  int main(int argc, char *argv[])
/*=========================================================*/
{   int i, r;
 // les gestionnaires de signaux doivent être installés
 // AVANT tout fork(). Sans cela, un fils pourrait se terminer
 // prématurement suite à la réception de SIGUSR1 ou SIGUSR2
 // alors qu'il n'a pas encore installé les gestionnaires

      signal (VAS_Y_P2,  OK_D); // Quand P2 recoit VAS_Y_P2,
                                // il peut exécuter l'action D
      signal (VAS_Y_P1,  OK_B); // Quand P1 recoit VAS_Y_P1,
                                // il peut exécuter l'action B

    P=getpid ();

    P1=fork();
      if (P1<0){
            perror("PB fork ");
            _exit (EXIT_FAILURE);
      } // if (P1<0)

      if (P1==0){ // ========= code de P1 ================
      // P1 ne peut connaitre le PID de son frère P2
      // qui sera créé APRES lui. C'est son père qui
      // relayera son signal vers P2

        P2=P;
        P1=getpid();

        action_A ();

        vas_y_P2 (); // P1 informe P2, via son père, que A est terminée

        synchro_avec_P2 ();// D de P2 est-elle terminée ??
                           // P1 attend que P2 ait terminée D
        action_B();
        exit (EXIT_SUCCESS);
      } // if (P1==0)

      P2=fork(); // P
      if (P2<0){
            perror("PB fork ");
            _exit (EXIT_FAILURE);
      } // if (P2<0)



      if (P2==0) {// ============= code de P2  ===============
       // P2 hérite de l'identité de son frère P1 qui a été
       // créé AVANT lui.
          P2=getpid();
```

```
        action_C ();

        synchro_avec_P1 (); // A de P1 est-elle terminée ??

        action_D();

        vas_y_P1 (); // P2 informe P1 que D est terminée
        exit (EXIT_SUCCESS);
      } // if (P2==0)

  // ============= code de P  ===============

    synchro_avec_P1 (); // Attente du signal venant de P1
                        // afin de le relayer vers P2

    vas_y_P2 (); // Relai du signal vers P2

    for (i=1; i<=2; i++) {
          r=wait(NULL);
          if (r<0) perror ("PB wait ");
    }
    printf ("FIN\n");
}// main
/* ============================================================

-bash-4.0$ ./a.out
P 9550: J'attends
          P1 9551: Execution action A
          P1 9551: vas_y_P2
P 9550: vas_y_P2
          P1 9551: J'attends
                     P2 9552: Execution action C
                     P2 9552: Je n'attends pas
                     P2 9552: Execution action D
                     P2 9552: vas_y_P1
          P1 9551: Execution action B
FIN

-bash-4.0$ ./a.out
          P1 9554: Execution action A
          P1 9554: vas_y_P2
          P1 9554: J'attends
                     P2 9555: Execution action C
                     P2 9555: J'attends
P 9553: J'attends
^C ICI un bloquage s'est produit !!
   Il est dû à l'utilisation de pause
   au lieu de sigsuspend
-bash-4.0$ ./a.out

P 9565: J'attends
                     P2 9567: Execution action C
                     P2 9567: J'attends
          P1 9566: Execution action A
          P1 9566: vas_y_P2
          P1 9566: J'attends
P 9565: vas_y_P2
                     P2 9567: Execution action D
                     P2 9567: vas_y_P1
          P1 9566: Execution action B
FIN
-bash-4.0$ ./a.out
          P1 9569: Execution action A
P 9568: J'attends
          P1 9569: vas_y_P2
P 9568: vas_y_P2
```



```
          P1 9569: J'attends
                     P2 9570: Execution action C
                     P2 9570: Je n'attends pas
                     P2 9570: Execution action D
                     P2 9570: vas_y_P1
          P1 9569: Execution action B
FIN

-bash-4.0$ ./a.out
P 9635: J'attends
          P1 9636: Execution action A
                     P2 9637: Execution action C
          P1 9636: vas_y_P2
                     P2 9637: J'attends
          P1 9636: J'attends
P 9635: vas_y_P2
                     P2 9637: Execution action D
                     P2 9637: vas_y_P1
          P1 9636: Execution action B
FIN

  ============================================================ */
```

ANNEXE

Droits d'accès

1. si ID user effectif du processus est 0 (superuser) l'accès est autorisé,
2. si ID user effectif du processus est égal à ID propriétaire du fichier :
   a. si le mode d'accès correspond aux droits d'accès propriétaire, l'accès est autorisé,
   b. sinon l'accès est refusé,
3. si ID groupe effectif du processus, ou l'un des IDs groupes supplémentaires, du processus est égal à ID groupe du fichier :
   a. si le mode d'accès correspond aux droits d'accès du groupe, l'accès est autorisé,
   b. sinon l'accès est refusé,
4. si le mode d'accès correspond aux droits d'accès des autres, l'accès est autorisé, sinon l'accès est refusé.