TP n°3 - Correction

Fonction *exec*

Pour toute question pendant le TP, commencez par consulter l'aide de `exec` en entrant la commande `man 3 exec`.

Exercice 1 [Rappel d'UNIX : gestion des processus]

Se familiariser avec les différentes options de la commande `ps` en lisant `man ps`.

1. Quel est le processus de pid 1 ?
2. Lancer le programme ci-dessous avec les arguments 10 20. Tapez ps -la dans un autre terminal avant la fin du père, avant la fin du fils. Quels sont les ppid du pere et du fils ? Donnez une explication.
3. Lancer le programme ci-dessous avec les arguments 10 0. Tapez ps -la dans un autre terminal avant la fin du père. Que constatez-vous ?

début src/ex1.c

```
#include <unistd.h> /* necessaire pour les fonctions exec */
#include <sys/types.h> /* necessaire pour la fonction fork */
#include <unistd.h> /* necessaire pour la fonction fork */
#include <stdio.h> /* necessaire pour la fonction perror */

int main(int argc, char * argv[]) {

  pid_t pid;
  int attente_fils,attente_pere;
  if(argc != 3)
    perror("usage: ex1 n m\n");

   attente_pere = atoi(argv[1]);
   attente_fils = atoi(argv[2]);

  switch(pid=fork()) {
  case -1:
    perror("fork error");
    break;
  case 0:
    sleep(attente_fils);
    printf("fils attente finie\n");
    break;
  default:
    sleep(attente_pere);
    printf("pere attente finie\n");
    break;
```

```
  }

}
```

fin src/ex1.c

Correction :

1. Utiliser les options `l a x` de `ps` pour pouvoir voir le processus `init`.
2. Voir que le père du programme est le terminal. Sentir ce qui se passe quand le père meurt avant le fils (init devient le père).
3. Voir ce qui se passe quand le fils meurt avant le père. Processus zombies (state Z).

Sentir pourquoi, il faudra utiliser les fonctions `wait`.

Exercice 2 [Comportement de `exec`]

1. Faites `exec ps`. Que se passe-t-il ? Donnez une explication.
2. Vérifiez votre explication en étudiant la commande shell `exec sh`. (Quel est le `pid` de ce shell ?)

Correction : Comprendre le fonctionnement d'`exec` et qu'il faudra le coupler avec `fork`.

Exercice 3 [Path et variables d'environnement]

1. Créez dans `progs` un programme `affichez` qui affiche une chaîne de caractères passée en argument.
2. Placez vous dans `tp3` et tapez `affichez salut`. Que se passe-t-il ?
3. Comment faire pour que cette commande exécute le programme `affichez` ?
   Pour vous aidez, demandez vous :
   Que donne la commande shell `which ls` ?
   Que donne la commande shell `echo $PATH` ?

Correction : Comprendre comment UNIX trouve les programmes qu'on lui demande d'exécuter.
Changer des variables d'environnement : PATH=$PATH :.../tp3/progs/

Exercice 4 Ecrire un programme `prog1` qui crée un processus fils qui exécute `affichez` avec l'argument `salut`. On utilisera la fonction `execl`.

Correction : Remarque : ne pas oublier le nom du programme dans la liste des arguments.

début src/prog1.c

```
#include <unistd.h> /* necessaire pour les fonctions exec */
#include <sys/types.h> /* necessaire pour la fonction fork */
#include <unistd.h> /* necessaire pour la fonction fork */
#include <stdio.h> /* necessaire pour la fonction perror */


int main() {
```

```
  pid_t pid;

  if((pid = fork()) < 0)
    perror("fork error");
  else if(pid == 0) {
    if(execl("/export/giroire/tp3/progs/affichez","affichez","salut",(char *) 0) < 0)
      perror("execl error");
  }
}
```

fin src/prog1.c

Exercice 5 Ecrire un programme `prog2` qui crée un processus fils qui a un environnement à seulement deux variables PATH=/src et USER=unknown. Pour vérification, il affichera ses arguments et ses variables d'environnement. On utilisera cette fois la fonction `execve`.

On s'inspirera du programme ci-dessous qui affiche les valeurs des variables d'environnement de votre machine.

début src/env.c

```
#include <stdio.h> /* necessaire pour la fonction perror */

int main() {

  char ** ptr;
  extern char **environ;

  for(ptr = environ;*ptr != 0;ptr++)
    printf("%s\n",*ptr);

}
```

fin src/env.c

Correction :

début src/argenv.c

```
#include <stdio.h> /* necessaire pour la fonction perror */


int main(int argc, char * argv[]) {

  char ** ptr;
  extern char **environ;
  int i;

  printf("\nArguments : \n");
  for(i=0;i<argc;i++)
    printf("%s\n",argv[i]);

  printf("\nVariables d'environnements : \n");
```

```
  for(ptr = environ;*ptr != 0;ptr++)
    printf("%s\n",*ptr);

}
```

fin src/argenv.c

début src/prog2.c

```
#include <unistd.h> /* necessaire pour les fonctions exec */
#include <sys/types.h> /* necessaire pour la fonction fork */
#include <unistd.h> /* necessaire pour la fonction fork */
#include <stdio.h> /* necessaire pour la fonction perror */

char * env_init[] = {"PATH=/export/giroire/Monitorat/Systeme/Td3-4/tp3/progs/","LOGIN=unknown",NULL};


int main() {

  pid_t pid;
  char * arguments[] = {"argenv",(char *) 0};

  if((pid = fork()) < 0)
    perror("fork error");
  else if(pid == 0) {
    if(execve("/export/giroire/Monitorat/Systeme/Td3-4/tp3/progs/argenv",arguments,env_init) < 0)
      perror("execl error");
  }

}
```

fin src/prog2.c

Exercice 6 [Fonction `wait`]

Ecrire un programme qui crée 2 processus l'un faisant la commande `ls -l`, l'autre `ps -l`. Le père devra attendre la fin de ses deux fils et afficher quel a été le premier processus à terminer.

Correction : début src/ex6.c

```
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char * argv[]) {

  pid_t pid1,pid2,pid_premier;
  int status;

  switch(pid1=fork()) {
  case -1:
    perror("fork error");
```

```
    break;
  case 0:
    execlp("ls","ls",(char *) 0);
    break;
  default:
    switch(pid2=fork()) {
    case -1:
      perror("fork error");
      break;
    case 0:
      execlp("ps","ps",(char *) 0);
      break;
    default:
    break;
  }
    break;
  }

  pid_premier = wait(&status);
  wait(&status);

  printf("Premier processus a finir : %d\n", (pid_premier==pid1)?1:2);


}
```

fin src/ex6.c