Laboratoire

Process

Jaumain J-C

révision mba - octobre 2018

Table des matières

Chapitre 1

fork

1.1 Process - LaboProcess 01-01 - fork clonage et adoption

Titre :	Process - LaboProcess 01-01 - fork clonage et adoption
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

1.1.1 Énoncé

Écrivez un programme qui se clone et ensuite affiche son pid, le contenu et l'adresse d'une variable locale. Vérifiez l'ordre d'exécution des process, le contenu et l'adresse de la variable. Inversez l'ordre des affichages en utilisant la fonction sleep. Ce résultat est-il déterministe ? Autrement dit, obtiendra-t-on toujours ces mêmes valeurs ?

1.1.2 Une solution

```
/*
NOM      : Fork.c
CLASSE   : Process - LaboProcess 01-01
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main ( )
{  int var,r;
   var=6;
   printf("Avant le fork, pour le père %d, l'adresse %p contient %d\n", getpid(),(void*)&var,var);
   if ((r=fork())==0){
      printf ("Voici %d, fils de %d\n", getpid(), getppid());
      printf("pour le fils %d, l'adresse %p contient %d\n", getpid(),(void*)&var,var);
      var=19;
      //sleep(3);
      printf("pour le fils %d, l'adresse %p contient %d\n", getpid(),(void*)&var,var);
      exit(0);
   }
   printf("Après le fork, pour le père %d, l'adresse %p contient %d\n", getpid(),(void*)&var,var);
   var=13;
   //sleep(3);
   printf("Après le fork, pour le père %d, l'adresse %p contient %d\n", getpid(),(void*)&var,var);
   wait(0);
   exit(0);
}
```

1.1.3 Commentaires

— Une fois le process fils crée, les deux process sont indépendants.
— Chaque process a son propre espace d'adressage, il n'y a pas de partage de la variable locale.
— L'adresse de la variable est une adresse relative à l'espace d'adresseage de chaque processus, les espaces sont séparés physiquement.
— Le shell attend que son fils soit terminé avant d'envoyer le prompt, il n'attend pas son petit-fils. Selon l'ordonnancement, il se peut qu'on ne voit pas l'affichage du prompt, mais ce dernier a bien eu lieu. Pour obtenir un affichage de prompt tardif, on peut ajouter sleep(1) dans le code du fils (cela dépendra du nombre d'autres processus prioritaires dans le système).
— L'appel système exit est indispensable chez le fils, sinon le process fils continue et affiche encore deux lignes de plus.
— L'ordre des impressions n'est pas garanti.

1.1.4 En roue libre

— Vérifiez le comportement pour les variables de classe d'allocation statique et dynamique.
— Adaptez cet exemple, en illustrant l'adoption par init.

Chapitre 2

wait

2.1 Process - LaboProcess 02-01 - wait4

Titre :	Process - LaboProcess 02-01 - wait4
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

2.1.1 Énoncé

Écrivez un programme Wait4 qui crée un process fils.
Si l'argument est u, ce programme crée un process fils qui effectue le calcul de 10^6 sinus de nombre aléatoires.
Si l'argument est s, ce programme crée un process fils qui effectue l'impression de 10^6 caractères a
Le père attend la fin de son fils et affiche les ressources consommées par son fils.

2.1.2 Une solution

```
/*
NOM      : Wait4.c
CLASSE   : Process - LaboProcess 02-01
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<sys/time.h>
#include<sys/types.h>
#include<sys/resource.h>
#include<sys/wait.h>
#include<fcntl.h>
#include<sys/stat.h>
#include<unistd.h>
int main(int argc, char * argv[])
{  pid_t pid;
   int status,i,h;
   struct rusage usage;
   if (argv[1][0]=='u')
   {  if ((pid = fork()) == 0)
      {  srand(getpid());
         printf("pid du fils =%u\n", getpid());
               for(i=0; i < 10000000; i++)
            sin(rand());
               exit(0);
      }
   }
   if (argv[1][0]=='s')
   {  if ((pid = fork()) == 0)
      {  printf("pid du fils =%u\n", getpid());
```

```
            h=open("/dev/null",O_RDWR);
                  for(i=0; i < 10000000; i++)
               write(h,"a",1);
            close(h);
                  exit(0);
         }
      }
      if ((wait4(pid, &status, 0, &usage)) > 0 )
      {  printf("Temps utilisateur : %ld s, %ld micro sec\n",
                  usage.ru_utime.tv_sec, usage.ru_utime.tv_usec);
               printf("Temps en mode noyau : %ld s, %ld micro sec\n",
                  usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
      }
      exit(0);
}
```

2.1.3 Commentaires

— L'argument u (pour utilisateur) demande beaucoup de ressources utilisateur sans faire appel au S.E
— L'argument s (pour système) demande beaucoup de ressources utilisateur et fait appel au S.E avec l'appel système write.
— On a choisi ici d'écrire dans un périphérique /dev/null qui reçoit les informations mais ne les utilise pas. Ceci évite un affichage de a à l'écran qui prend beaucoup de temps.
— La structure rusage peut être lue via "man getrusage".
— Le préfixe time permet de demander au shell la consommation d'une commande (man time).

2.2 Process - LaboProcess 02-02 - zombies

Titre :	Process - LaboProcess 02-02 - zombies
Support :	OS 42.3 Leap Installation Classique
Date :	02/2016

2.2.1 Énoncé

Écrivez un programme qui crée un zombie, affiche ensuite cet état zombie à l'aide de la fonction system, élimine ce zombie et affiche les process en cours (sans zombie).

2.2.2 Une solution

```
/*
NOM      : Zombie.c
CLASSE   : Process - LaboProcess 02-02
#OBJET   : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
int main(int argc, char * argv[])
{  char spid[100];
   int r;
   if ((r=fork()) == 0)
   {  exit(0);
   }
   printf("Le process fils qui devient zombie est le numéro %d\n",r);
   printf("Résultat affiché par ps : table des process\n\n");
   sleep(1);
   sprintf(spid,"ps -o ppid,pid,state,command"); // juste pour montrer un appel à sprintf ;-)
   system(spid);
   printf("\nJ'appelle wait, une bonne façon d'éliminer un zombie\n\n");
   sleep(1);
   wait(0);
   sleep(1);
   printf("Résultat affiché par ps après le wait : table des process\n\n");
   system(spid);
   sleep(1);
   exit(0);
}
```

2.2.3 Commentaires

— sprintf est la façon propre de transformer un nombre en chaîne de caractère. Écrivez un printf normal, ensuite ajoutez la chaîne destination pré allouée comme premier argument de sprintf.
— La fonction c system crée un shell et exécute la chaîne comme seule commande de ce shell.
— l'élimination du zombie est provoquée par l'appel système wait. Un kill n'a pas d'effet sur un Zombie, on ne tue pas un fantôme !

2.2.4 En roue libre

Adaptez ce code pour éliminer le zombie grâce à la technique du double fork qui permet de forcer l'adoption immédiate par le process 1
Vérifiez qu'il n'y a pas de création de zombies dans ce cas.

Chapitre 3

exec

3.1 Process - LaboProcess 03-01 - exec

Titre :	Process - LaboProcess 03-01 - exec
Support :	OS 42.3 Leap Installation Classique
Date :	02/2015

3.1.1 Énoncé

Écrivez un programme en C qui réalise la même chose que

gcc aff.c -o aff && ./aff Message

Le programme aff affiche le premier argument passé (Message)
Si aff.c contient une erreur de compilation, un message d'erreur est affiché.

3.1.2 Une solution

```
/*
NOM      : cg.c
CLASSE   : Process - LaboProcess 03-01
#OBJET   : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
int main()
{  int j;
   if (fork()==0)
   {  execlp("gcc","gcc","aff.c","-o","aff",0);
      printf("Erreur : gcc non trouvé\n");
      exit(1);
   }
   wait(&j);
   if (j != 0)
   {  printf("Erreur de compilation du programme aff.c\n");
      exit(1);
   }
   execl("./aff","aff","Message",0);
   printf("Ici, vous pouvez raconter votre vie, cela ne sera jamais affiché\n");
   exit(0);
}
```

```
/*
NOM      : aff.c
```

```
/*
CLASSE   : Process - LaboProcess 03-01
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{  printf("%s\n",argv[1]);
   exit(0);
}
```

3.1.3 Commentaires

— exec est le loader qui permet de charger un programme exécutable en mémoire.
— exec REMPLACE le process courant par le programme exécutable passé comme premier argument. Le process courant est entièrement détruit. Ceci justifie le fork et l'absence de test après exec gcc pour afficher le message d'erreur.
— gcc, comme toute commande, se termine par exit(0) si il n'y a pas d'erreur. C'est une convention.
— l'opérateur && du bash est un AND. La commande qui suit le && est exécutée ssi la première se termine par un exit(0) qui veut dire "tout s'est bien passé"

3.2 Process - LaboProcess 03-02 - exec et sécurité

Titre :	Process - LaboProcess 03-02 - exec et sécurité
Support :	OS 42.3 Leap Installation Classique
Date :	02/2015

3.2.1 Énoncé

Créez un fichier appelé Confidentiel qui contient le mot 'CONFIDENTIEL'. Écrivez un programme Conf qui affiche le contenu du fichier appelé Confidentiel en chargeant l'exécutable de la commande cat. Modifiez les droits de Confidentiel et de Conf de telle façon que personne d'autre que vous ne puisse afficher le contenu du fichier appelé Confidentiel via la commande cat, mais que tout le monde puisse afficher le contenu en exécutant le programme Conf.

3.2.2 Une solution

```
/*
NOM      : Conf.c
CLASSE   : FS - LaboFS 03-02
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main ()
{  printf ("User %d runs as user %d\n", getuid(), geteuid());
   execlp("cat","cat","Confidentiel",0);
   exit(0);
}
```

3.2.3 Commentaires

. Cette solution est dangereuse. exec et SUID ne vont pas bien ensemble

. Un utilisateur user1 peut se faire passer pour l'utilisateur propriétaire pour d'autres commandes que cat en utilisant le programme Conf.

Chapitre 4

shell

4.1 Process - LaboProcess 04-01 - shell simple

Titre :	Process - LaboProcess 04-01 - shell simple
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

4.1.1 Énoncé

Écrire un shell simple capable d'exécuter n'importe quelle commande externe avec un nombre d'argument quelconque, et la commande interne exit.

4.1.2 Une solution

```
/*
NOM      : Shell.c
CLASSE   : FS - LaboProcess 04-01
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <unistd.h>
int main ()
{  int i;
   char ligne[257];
   char *tokens[100];
   char errmsg[200];
   printf("$ ");
   fgets(ligne,256,stdin);
   while (strcmp(ligne,"exit\n"))
   {  i=0;
      tokens[i]=strtok(ligne," \n");
      while (tokens[i] != NULL) tokens[++i]=strtok(NULL," \n");
      if (fork()==0){
          execvp(tokens[0],tokens);
          sprintf (errmsg, "exec <%s> :", tokens[0]);
          perror(errmsg);
      }
      wait(0);
      printf("$ ");
      fgets(ligne,256,stdin);
   }
   exit(0);
}
```

4.1.3 Commentaires

— Le message 'commande invalide' ne demande pas de if. Si exec a trouvé l'exécutable, il écrase le process courant.
— Le fork clone le shell courant en un deuxième shell. Ce fils est écrasé par un exécutable si exec le trouve. Sinon, le fils reste un shell. Si on oublie de terminer le fils avec exit(), celui-ci reste un shell et deux shell s'exécutent. L'utilisateur devra alors taper deux fois la commande interne exit pour en sortir.
— Le script Demo contient le symbole «. Ce n'est pas une indirection mais définit le symbole de fin de donnée pour le process.

4.1.4 En roue libre

— Corrigez ce shell simple : votre shell doit se comporter correctement si l'utilisateur introduit le nom d'un exécutable qui n'existe pas.

4.2 Process - LaboProcess 04-02 - shell et background

Titre :	Process - LaboProcess 04-02 - shell et background
Support :	OS 42.3 Leap Installation Classique
Date :	02/2015

4.2.1 Énoncé

Modifier le shell simple pour y ajouter la notion de background.

4.2.2 Une solution

```
/*
NOM      : ShellBack.c
CLASSE   : FS - LaboProcess 04-02
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
#BUGS : Pour simplifier, on impose que le symbole & soit le dernier argument sur la ligne de commande (précédé par un espace).
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>
#include <sys/types.h>
#include <sys/wait.h>
int main ()
{  int i,r;
   bool bg;
   char *tokens[100];
   char errmsg[200];
   char *ligne;          // pour changer ;-)
   ligne=(char*)malloc(257);
   printf("$ ");
   fgets(ligne,256,stdin);
   while (strcmp(ligne,"exit\n"))
   {  i=0;
      bg=false;
      tokens[i]=strtok(ligne," \n");
      while (tokens[i] != NULL) tokens[++i]=strtok(NULL," \n");
      if ((i>0) && (strcmp(tokens[i-1],"&")==0))
      {  bg=true;
         tokens[i-1]=0;
      }
      if ((r=fork())==0)
      {  execvp(tokens[0],tokens);
         sprintf (errmsg, "exec <%s> :", tokens[0]);
            perror(errmsg);
      }
      if (!bg) waitpid(r,0,0);
      // sinon attention zombie
      printf("$ ");
      fgets(ligne,256,stdin);
   }
   free (ligne);
   exit(0);
}
```

4.2.3 Commentaires

- bg est mis à 0 à chaque passage et vaut 1 si le dernier mot de la ligne est &
- Le test i>0 empêche que la lecture de tokens[i-1] ne provoque une erreur de segmentation si la ligne est vide.
- tokens[i-1] est remis à 0 pour que le symbole & ne soit pas passé à la commande.
- Un process en bg => pas de wait => le process reste zombie tant que le shell ne se termine pas. Ce problème sera réglé lors des l'étude du chapitre suivant (IPC, les signaux).

- waitpid et pas wait qui est insuffisant ici. Si vous utilisez un simple wait :
  - Vous lancez un process court en bg ;
  - Le process en bg se termine, vous gardez ce process fils à l'état zombie ;
  - Vous lancez une commande longue en fg ;
  - wait (0) attend la fin d'un fils quelconque, le zombie dans ce cas puisqu'il est fini.
  - Le zombie est libéré, le père débloqué affiche le prompt
  - On aurait une inversion de comportement : le process en fg s'exécute comme si il était en bg.

4.3 Process - LaboProcess 04-03 - shell et redirections

Titre :	Process - LaboProcess 04-03 - shell et redirections
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

4.3.1 Énoncé

Modifier le shell simple pour y ajouter la notion de redirection >.

4.3.2 Une solution

```
/*
NOM      : ShellBack.c
CLASSE   : FS - LaboProcess 04-03
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
#BUGS : Pour simplifier, on impose que le symbole > soit l'avant dernier mot de la ligne de
#  : commande, le dernier mot étant le nom du fichier.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
int main ()
{  int i,h;
   char * ligne;
   char *tokens[100];
   ligne=(char*)malloc(300);
   printf("$ ");
   fgets(ligne,256,stdin);
   while (strcmp(ligne,"exit\n"))
   {  i=0;
      tokens[i]=strtok(ligne," \n");
      while (tokens[i] != NULL) tokens[++i]=strtok(NULL," \n");
      if (fork()==0)
      {  if ((i>1) && (strcmp(tokens[i-2],">")==0))
         {  h=open(tokens[i-1],O_WRONLY|O_CREAT|O_TRUNC,0644);
            dup2(h,1);
            close(h);
            tokens[i-2]=0;
         }
         execvp(tokens[0],tokens);
         perror("exec");
      }
      wait(0);
      printf("$ ");
      fgets(ligne,256,stdin);
   }
   free (ligne);
   exit(0);
}
```

4.3.3 Commentaires

— open doit être appelé pour créer le fichier 'fichier'. Il ne faut pas oublier d'ajouter les droits sinon l'appel système prend le contenu de EDX pour les donner ce qui donne n'importe quoi !
— 0644, le 0 devant signifie 'en octal'.

4.3.4 En roue libre

Adaptez cette solution pour la redirection de l'entrée standard "<

Chapitre 5

pipe

5.1 Process - LaboProcess 05-01 - pipe et shell

Titre :	Process - LaboProcess 05-01 - pipe et shell
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

5.1.1 Énoncé

Écrivez un shell simplifié, seulement capable d'exécuter toujours la commande `ps aux | grep root | wc -l` ou la commande exit pour terminer. Ce shell ne reconnaît que la commande exit.

5.1.2 Une solution

```
/*
NOM      : PipeShell.c
CLASSE   : Process - LaboProcess 05-02
#OBJET  : réservé au Makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main ()
{  int fp1[2],fp2[2];
   char ligne[257];
   printf("Commande ?] ");
   fgets(ligne,256,stdin);
   while (strcmp(ligne,"exit\n"))
   {     pipe(fp1); pipe(fp2);
      if (fork()==0)
      {  dup2(fp1[1],1);
         close(fp1[0]);close(fp1[1]);close(fp2[0]);close(fp2[1]);
         execlp("ps","ps","aux",0);
      }
      if (fork()==0)
      {  dup2(fp1[0],0);
         dup2(fp2[1],1);
         close(fp1[0]);close(fp1[1]);close(fp2[0]);close(fp2[1]);
         execlp("grep","grep","root",0);
      }
      if (fork()==0)
      {  dup2(fp2[0],0);
         close(fp1[0]);close(fp1[1]);close(fp2[0]);close(fp2[1]);
         execlp("wc","wc","-l",0);
```

```
        }
        close(fp1[0]);close(fp1[1]);close(fp2[0]);close(fp2[1]);
        while(wait(0)>0);
        printf("Commande ?] ");
        fgets(ligne,256,stdin);
    }
    exit(0);
}
```

5.1.3 Commentaires

— Une ligne avec 3 commandes
— Le shell, 3 fork() et 2 pipe() = 16 close !
— N'oublier aucun close, y compris chez le père.
— Pas de wait entre les fork() : ces process doivent s'exécuter 'en même temps'.
— Les close() avant le wait() chez le père sinon deadlock !

5.2 Process - LaboProcess 05-02 - pipe et shell - à corriger

Titre :	Process - LaboProcess 05-02 - pipe et shell - à corriger
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

5.2.1 Énoncé

Ce programme est erroné. Il devrait exécuter la commande `cat /etc/passwd | cut -f6 -d ':'| sort` ou la commande exit pour terminer.
Vérifiez-en le comportement et expliquez le. Terminez en corrigeant les erreurs.

5.2.2 Une solution

```
/*
NOM      : PipeShellErr.c
CLASSE   : Process - LaboProcess 0502
#OBJET  : réservé au Makefile
AUTEUR   : M Bastreghi, 09/2017
*/
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

int main ()
{  int fp1[2],fp2[2];
   char ligne[257];
   printf("Commande ?] ");
   fgets(ligne,256,stdin);
   while (strcmp(ligne,"exit\n"))
   {     pipe(fp1); pipe(fp2);
      if (fork()==0)
      {  dup2(fp1[1],1);
         execlp("cat","cat","/etc/passwd",0);
      }
      if (fork()==0)
      {  dup2(fp1[0],0);
         dup2(fp2[1],1);
         execlp("cut","cut","-f6", "-d" ":", 0);
      }
      if (fork()==0)
      {  dup2(fp2[0],0);
         execlp("sort","sort",0);
      }
      close(fp1[0]);close(fp1[1]);close(fp2[0]);close(fp2[1]);
      while(wait(0)>0);
      printf("Commande ?] ");
      fgets(ligne,256,stdin);
   }
   exit(0);
}
```

5.2.3 Commentaires

— Une ligne avec 3 commandes et 2 pipes –> 3 fork() et 2 pipe().
— Pas de wait entre les fork() : ces process doivent s'exécuter 'en même temps'.
— N'oublier aucun close, y compris chez le père.
— Les close() avant le wait() chez le père sinon deadlock !

Chapitre 6

signal

6.1 Process - LaboProcess 06-01 - trap du ctrl-c

Titre :	Process - LaboProcess 06-01 - trap du ctrl-c
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

6.1.1 Énoncé

Écrivez un programme qui affiche "aie 1 fois", "aie 2 fois",... "aie n fois", "j'ai compris" lorsqu'on pousse sur ctrl-c. Le nombre n est donné comme premier argument (de 0 à n) et vaut 0 par défaut.

6.1.2 Une solution

```
/*
NOM      : Sigaction.c
CLASSE   : Process - LaboProcess 06-01
#OBJET  : réservé au Makefile
AUTEUR   : M. Bastreghi 11/2017
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#define __USE_POSIX
#define __USE_POSIX199309
#define __USE_UNIX98

#include <signal.h>

void trapc(int sig);
static int iter=0,cpt,max;
static char lettre;
static struct sigaction act;


int main (int argc, char * argv[])
{
   if (argc == 2)  {
      max=atoi(argv[1]);
      act.sa_handler = trapc;
      sigaction (SIGINT,&act,NULL);

   }
   for (cpt=0;cpt>=0;cpt++)
   {  lettre='A'+(cpt%26);
      write(1,&lettre,1);
      sleep(1);
   }
```

```
   exit(0);
}

void trapc(int sig)
{  if (iter >= max)
   {  printf("    J'ai compris; process stoppé ! \n");
      exit(0);
   }
   else  printf("    aie %d fois\n",++iter);
}
```

6.1.3 Commentaires

- Pour obtenir de l'aide sur l'utilisation de l'appel système signal : man 2 sigaction
- Pour obtenir la liste des signaux : man 7 signal
- Remarquer la boucle infinie dans le programme principal pour laisser à l'utilisateur le temps d'entrer Ctrl-c.

6.1.4 En roue libre

Le traitement du signal sera fait une seule fois. On se passera de compteur.

6.2 Process - LaboProcess 06-02 - trap de tous les signaux

Titre :	Process - LaboProcess 06-02 - trap de tous les signaux
Support :	OS 42.3 Leap Installation Classique
Date :	07/2011

6.2.1 Énoncé

Écrivez un programme qui traite tous les signaux non temps réel. Observez le comportement de ce programme en lui envoyant plusieurs signaux et CTRL-C.

6.2.2 Une solution

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

#define __USE_POSIX
#define __USE_POSIX199309
#include <signal.h>

void trapall(int sig, siginfo_t * pinfo, void * pucontext);
struct sigaction act;

int main (int argc, char * argv[])
{
   int noSig;
   int retour;
   char errmessage [256];

   printf ("Bonjour, je suis %d\n", getpid());
   act.sa_flags = SA_SIGINFO;
   act.sa_sigaction = trapall;

   for (noSig=1; noSig<32; noSig++){
      char * err;
      if ((retour = sigaction (noSig,&act,NULL)) < 0) {
         err = strerror(errno);
         sprintf (errmessage, "sigaction %d : ", noSig);
         strncat (errmessage, err,255);
         fprintf (stderr, errmessage );
         fprintf (stderr,"\n");
      }
   }

   while (1) pause();
}

void trapall(int sig, siginfo_t * pinfo, void * pucontext){
   printf("reçu le signal=%d\n", pinfo->si_signo);
}
```

6.2.3 Commentaires

— Les signaux SIGKILL et SIGSTP ne peuvent être traités.
— L'utilisation de l'appel système pause permet d'éviter l'attente active.

6.2.4 En roue libre

— Modifiez Le comportement du programme de manière à ce qu'un signal traité une fois soit ensuite ignoré en utilisant SIG_IGN.

Chapitre 7

Exercices

+ Process002 : Déterminer les valeurs de glob et loc affichées par ce programme. Justifiez votre réponse.

```
int glob=45;
int main()
{ int loc=78; printf("< %d %d\n",glob,loc);
   if (fork()= =0) { glob=65; loc=32; exit(0);}
   wait(0);
   printf("> %d %d\n",glob,loc);
   exit(0);
}
```

+ Process004 : fork( ) dédouble-t'il la mémoire pointée par un pointeur ou uniquement le pointeur ? Imaginez un exemple qui prouve votre réponse. La valeur du pointeur est-elle la même chez le fils et le père ? Justifiez votre réponse.
+ Process019 : Écrivez un shell simplifié seulement capable d'exécuter toujours la commande `ls -ail >x 2>x` ou `exit`. Peu importe ce que vous écrirez comme commandes au clavier.
+ Process025 : On vous donne un shell simple. Modifiez-le afin d'y ajouter l'interprétation du caractère * (isolé).
+ Process027 : On vous donne un shell simple en annexe. Comment réagit ce shell si vous tapez une commande qui n'existe pas ? Expliquez ce comportement et apportez une solution.
+ Process036 : Vous trouvez un programme suid root rwsr-xr-x qui contient la ligne
  `execlp("ps","ps","aux",0)`
  Décrivez comment vous pouvez devenir administrateur du système sans en connaître le passwd. Effectuez une démonstration de cette description.
+ Process045 : Écrivez un programme qui gère les zombies. A chaque boucle, ce programme lit un message au clavier. Si ce message vaut
  — c : le programme crée 2 nouveaux zombies et affiche la liste des zombies 'actuels' via la commande ps.
  — d nnnnn : le programme élimine le zombie de pid nnnnn et affiche la liste des zombies 'actuels' via la commande ps.
  — q : le programme s'arrête.
  Que deviennent les éventuels zombies restants quand vous quittez ce programme ?
+ Process049 : Écrivez un programme qui simule un shell seulement capable d'exécuter toujours la commande `cd ;ls»out` ou `exit`. Peu importe ce que vous écrirez comme commandes au clavier.
+ Process053 : Écrivez shell simplifié seulement capable d'exécuter toujours la commande `cd ; (cd .. ;ls) ; ls` ou `exit`. Peu importe ce que vous écrirez comme commandes au clavier. Veillez à programmer le ; et les parenthèses.

+ Process055 : Écrivez un shell simplifié seulement capable d'exécuter toujours la commande `mkdir brol && (cd brol ;>f)` ou `exit`. Peu importe ce que vous écrirez comme commandes au clavier. Veillez à programmer le ; et les parenthèses.
+ Process2_004 : Écrivez un shell simplifié seulement capable d'exécuter toujours la commande `ls -ail | cat > x` ou `exit`. Peu importe ce que vous écrirez comme commandes au clavier.
+ Process2_009 : Réécrivez la commande sleep en c et baptisez-le Msleep. Elle utilisera les signaux (SIGALRM) et les appels alarm(n) qui envoie le signal SIGALRM après n secondes et pause() qui bloque le processus jusqu'à l'arrivée d'un signal.
+ Process2_012 : Écrire un programme c qui permet d'afficher de façon continue le dernier caractère introduit au clavier. (l'appel read() est bloquant !). Songez à communiquer avec un processus fils via un pipe et un signal.
+ Process2_025 : Testez le comportement d'un programme qui intercepte les signaux et affiche un message en clair. Par exemple : signal SIGSEGV reçu du au fait que le programme tente d'accéder à des données d'un pointeur non initialisé. Faites de même avec au moins 3 signaux différents.
+ Process2_027 : Écrivez un process qui envoie une série de signaux à un autre process. Ces signaux sont différents et envoyés rapidement, dans un ordre quelconque. Montrez que le process qui reçoit les signaux ne les reçoit pas nécessairement dans le même ordre. Expliquez.
+ Process2_039 : Écrivez un programme qui affiche 10 fois "bonjour n" où n va de 0 à 9, à distance de 3 secondes. Votre programme n'utilisera pas l'appel système sleep.
+ Process2_054(*) : Écrivez un shell simplifié seulement capable d'exécuter toujours les commandes `ls f 2>err | cat ; cat err` ou `exit`. Peu importe ce que vous écrirez comme commandes au clavier.