TP 6-7

Signaux

Exercice 1 (Signaux POSIX)**.**
Expliquer le fonctionnement du programme suivant. Donner un exemple d'affichage.

Comment aurait-on pu écrire ce programme avec les appels systèmes simplifés (`signal`*, etc.) ?*

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

struct sigaction action;

void hand_sigusr1(int sig){
     printf("(de %d) Signal SIGUSR1 recu\n", getpid());
     exit(0);
}

int main() {
  pid_t pid;
  int i = 0;
  int status;

  action.sa_handler=hand_sigusr1;
  sigaction(SIGUSR1,&action,NULL);

  if((pid=fork())==0){
    printf("Je suis le fils de PID %d\n", getpid());
    /* processus fils bouclant */
    while(1) { i = 0; };
  }
  printf("Je suis le pere de PID %d\n", getpid());

  if(kill(pid,0)==-1){
    printf("(de %d) Fils %d inexistant\n", getpid(), pid);
  }
  else{
    printf("(de %d) Envoi du signal SIGUSR1 au processus %d\n", getpid (), pid);
    kill(pid,SIGUSR1);
  }
  pid=waitpid(pid,&status,0);
  printf("(de %d) Status du fils %d : %d\n",getpid(),pid,status);
}
```

Manipulation 1 (Assez attendu ?)**.**
La suite de Syracuse est définie par :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} \text{ si } u_n \text{ est pair} \\ 3u_n + 1 \text{ sinon} \end{cases} \quad (1)$$

Une conjecture est que quelque soit le terme initial u_0 (non nul) de la suite, celle-ci finit par valoir 1 *(puis boucler sur* 4, 2, 1*). Étant donnée une valeur initiale u_0 de la suite, on appelle* temps de vol*, le plus petit indice k pour lequel $u_k = 1$ pour la première fois, et*altitude maximale *la valeur maximale prise par la suite durant ce temps de vol. Nous utilisons un programme calculant la suite de Syracuse pour chaque valeur initiale comme exemple d'utilisation possible des signaux.*

Pour simplifier, les variables liéées à la suite de Syracuse seront globales.

1. *Écrire une fonction* `void syracuse()` *qui étant donnée une valeur initiale calcule les termes de la suite de Syracuse jusqu'à ce qu'elle vaille* 1*.*
2. *Faire une boucle infinie pour $i = 1, 2, 3, \ldots$ qui fixe $u_0 = i$ et lance le calcul précédent.*

Le programme boucle indéfiniment sans rien afficher.

En utilisant l'interface simplifiée de gestion des signaux et alarmes (`signal`,`alarm`*) :*

3. *Faire en sorte qu'à la réception du signal* `SIGTSTP` *(Ctrl-Z), le programme affiche la valeur initiale courante, l'indice du dernier terme calculé et la valeur prise par la suite, sans stopper le programme.*
4. *Faire en sorte que Ctrl-C (*`SIGINT`*) termine le programme en affichant : la plus grande valeur initiale testée ; l'altitude maximale atteinte et la valeur initiale pour laquelle cette altitude maximale est atteinte ; le temps de vol maximal et la valeur initiale pour laquelle ce temps de vol a été atteint.*
5. *Faire en sorte que Ctrl-C ne termine pas le programme mais que retaper Ctrl-C dans les deux secondes termine le programme (utiliser une alarme).*

Bonus : même exercice en utilisant l'interface POSIX de gestion des signaux (`sigaction`*,* `sigalarm`*).*

Manipulation 2.
On souhaite faire un programme qui :

- *créé un tube et N processus fils ($N = 5$).*
- *Lorsqu'il reçoit le signal* `SIGUSR1` *le père écrit un nombre sur le tube et renvoie le signal* `SIGUSR2` *au fils qui a émis* `SIGUSR1`*.*
- *Chaque fils :*
  - *s'il n'a rien à faire émet le signal* `SIGUSR1` *au père.*
  - *s'il reçoit le signal* `SIGUSR2`*, lit le nombre dans le tube et exécute un calcul long sur ce nombre (on pourra simplement endormir le processus pendant 5 secondes).*

1. *Pouvez-vous utiliser la gestion simplifiée des signaux (*`signal`*, etc.) pour ce programme ou faut-il utiliser les signaux POSIX (*`sigaction`*, etc.) ?*
2. *Écrire une première version qui ne s'occupe que de l'envoie du signal* `SIGUSR1` *au père (et pas du tube et de l'autre signal). Faire afficher au père le nombre de signaux reçus. Tester votre programme. Tous les signaux sont-ils bien reçus ? Expliquer et proposer une solution.*

3. *Traiter le signal* `SIGUSR2`. *Tester.*

4. *Ajouter l'utilisation des tubes. Tester.*

5. *Comment, en fin de calcul un fils peut-il renvoyer un nombre au père ?*

Manipulation 3 (Pour aller plus loin : faire des blagues à gdb)**.**
Le debuggueur `gdb` *utilise l'opcode* `int3` *(*`0xCC`*) pour génèrer ses* breakpoints*, mais cet opcode génère un signal* `SIGTRAP` *lorsqu'il est utilisé dans un environnement d'exécution normal (ie hors debuggueur).*

Ceci peut être exploité pour modifier le comportement d'un programme lorsqu'il est exécuté par un debuggueur.

Si vous n'avez jamais utilisez de débuggueur regardez la page de man de `gdb` *(débugguage en ligne de commande), essayez quelques commandes sur un programme simple. Testez également* `kdbg` *(un débuggueur graphique, il y en a d'autres tels que* `ddd`*). Utilisez les* breakpoints*.*

Tester et expliquer le comportement des programmes suivants dans et hors débuggueur.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
void sighandler(int signal) {
  printf("Le pass est \"salut\"\n");
  exit(0);
}
int main(void) {
  signal(SIGTRAP,sighandler);
  __asm__("int3");
  printf("pris au piege\n");
  return EXIT_FAILURE;
}
```

```
#include <stdio.h>
#include <stdlib.h>

void foo(void) {
  printf("Salut\n");
}

int main(void) {
  unsigned long *i;
  unsigned long *min, *max;
  if ( (unsigned long *)foo <
       (unsigned long *)main ) {
    min = (unsigned long *)foo;
    max =  (unsigned long *)main;
  } else {
    max = (unsigned long *)foo;
    min =  (unsigned long *)main;
  }
  printf("%d\n",sizeof(long));
  for(i = min; i <  max; i++) {
    if(((*i & 0xff) == 0xCC) ||
       ((*i & 0xff00) == 0xCC) ||
       ((*i & 0xff0000) == 0xCC) ||
       ((*i & 0xff000000) == 0xCC) ) {
      printf("Breakpoint detecte \n");
      exit(EXIT_FAILURE);
    }
    foo();
    return 0;
  }
}
```

Corrigé

Correction de l'exercice 1.

Le processus père envoie un signal SIGUSR1 à son processus fils après avoir testé son existence (signal 0). Le fils se termine à la récéption de ce signal. Le père affiche alors le status de sortie du fils (marche pas !).

Correction de l'exercice 2.

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

long in, inmax, inkmax, inumax;
long long k, kmax;
long long u, umax;
int finir = 0;

void sig_interruption(int x){
  printf("\n");
  printf("Terme initial : %lu, syrac(%llu) = %llu\n", in, k, u);
  printf("C'est passionnant je continue !\n");
}

void infos(void) {
  printf("\n");
  printf("Plus grand terme initial : %ld\n", inmax);
  printf("Altitude maximale        : %lld (v.i. %ld)\n", umax, inumax);
  printf("Plus long temps de vol   : %lld (v.i. %ld)\n", kmax, inkmax);
}

void sig_fin(int sig) {
  if ( !finir ){
    infos();
    printf("Retapper rapidement Ctrl-C pour quitter\n");
    finir = 1;
    alarm(2);
  } else {
    infos();
    exit(0);
  }
}

void sig_alarme(int sig){
  finir = 0;
}

void syracuse(){
  while ( u > 1 ) {
    if (u % 2) u = 3*u + 1;
    else u = u/2;
    if (u > umax) {
```

```
      umax = u;
      inumax = in;
    }
    k++;
  }
}

int main () {
  u = -1;
  printf("Nb bits %d, max : %llu ie 10^%d\n",
 8 * sizeof u, u, 3 * (8 * sizeof u) / 10);
  signal(SIGTSTP, sig_interruption);
  signal(SIGALRM, sig_alarme);
  signal(SIGINT, sig_fin);
  for (in = 1; ; in++) {
    k = 0;
    u = in;
    syracuse();
    inmax = in;
    if (k > kmax) {
      kmax = k;
      inkmax = in;
    }
  }
}
```

Pour les signaux POSIX c'est la même chose... en plus compliqué ;-)

Correction de l'exercice 3.

Il faut apprendre le pid de l'émetteur d'un signal `SIGUSR1`. Accessoirement, il faudrait même vérifier que ce pid est bien celui d'un fils (on ne le fera pas).

Les signaux se superposent lorsqu'ils arrivent en même temps. Autrement dit, si le père est en train de traiter un signal `SIGUSR1` et qu'il en reçoit un ou plusieurs autres, ces derniers seront ignorés. On s'arrange donc pour que chaque fils émette le signal `SIGUSR1` régulièrement (tant qu'il est inoccupé).

Une solution (sauf de la dernière question qui nécessite un second tube et un autre jeu de signaux) :

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h> /* kill() getppid()*/
#include <signal.h>    /* kill(), sigaction() */
#include <unistd.h>    /* pipe() getppid() sleep() */

/* Remarque :
   info->si_pid donne toujours 0 sur mon mac os X pffff.
*/

#define N 10
```

```
int p[2];

int get_nombre() {
  static int x = 0;
  x++;
  return x * x + x + 1;
}

int gros_calcul(int x){
  int y;
  for (y = 0; ; y++) {
    if (y * y + y + 1 == x) return y;
  }
  sleep(5);
  return y;
}

void sig_pere_usr1(int sig, siginfo_t *info, void *ucont) {
  static int count = 0;
  int nombre;
  nombre = get_nombre();
  /*
    printf("SIGUSR1 recu du fils %d, total = %d, envoie de %d\n",
 info->si_pid, ++count, nombre);
  */
  write(p[1], &nombre, sizeof nombre);
  kill(info->si_pid, SIGUSR2);
}


void do_pere () {
  struct sigaction action_usr1;

  close(p[0]);

  action_usr1.sa_sigaction = sig_pere_usr1;
  action_usr1.sa_flags = SA_SIGINFO;
  sigaction(SIGUSR1, &action_usr1, NULL);

  while (1) sleep(1);
}


void sig_fils_usr2(int sig){
  int nombre;
  if (0 > read(p[0], &nombre, sizeof nombre)) perror("read fils");
  printf("%d a lu %d\n", getpid(), nombre);
  nombre = gros_calcul(nombre);
  printf("%d trouve %d\n", getpid(), nombre);
}

void do_fils () {
```

```
  struct sigaction action_usr2;

  close(p[1]);

  action_usr2.sa_handler = sig_fils_usr2;
  sigaction(SIGUSR2, &action_usr2, NULL);

  // printf("Fils %d\n", getpid());
  sleep(2);

  while(1) {
    kill(getppid(), SIGUSR1);
    sleep(2);
  }
}

int main () {
  int i;
  int pere = 1;
  if ( 0 > pipe(p)) perror("No pipe");
  for (i = 0; (i < N) && pere ; i++) {
    pere = fork();
    if (pere < 0) perror("fork");
  }
  if ( pere ) {
    do_pere();
  } else {
    do_fils();
  }
  exit(0);
}
```

Correction de l'exercice 4.

Il faut apprendre le pid de l'émetteur d'un signal SIGUSR1. Accessoirement, il faudrait même vérifier que ce pid est bien celui d'un fils (on ne le fera pas).

Les signaux se superposent lorsqu'ils arrivent en même temps. Autrement dit, si le père est en train de traiter un signal SIGUSR1 et qu'il en reçoit un ou plusieurs autres, ces derniers seront ignorés. On s'arrange donc pour que chaque fils émette le signal SIGUSR1 régulièrement (tant qu'il est inoccupé).

Une solution (sauf de la dernière question qui nécessite un second tube et un autre jeu de signaux) :

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h> /* kill() getppid()*/
#include <signal.h>    /* kill(), sigaction() */
#include <unistd.h>    /* pipe() getppid() sleep() */

/* Remarque :
```

```
   info->si_pid donne toujours 0 sur mon mac os X pffff.
*/

#define N 10

int p[2];

int get_nombre() {
  static int x = 0;
  x++;
  return x * x + x + 1;
}

int gros_calcul(int x){
  int y;
  for (y = 0; ; y++) {
    if (y * y + y + 1 == x) return y;
  }
  sleep(5);
  return y;
}

void sig_pere_usr1(int sig, siginfo_t *info, void *ucont) {
  static int count = 0;
  int nombre;
  nombre = get_nombre();
  /*
    printf("SIGUSR1 recu du fils %d, total = %d, envoie de %d\n",
 info->si_pid, ++count, nombre);
  */
  write(p[1], &nombre, sizeof nombre);
  kill(info->si_pid, SIGUSR2);
}


void do_pere () {
  struct sigaction action_usr1;

  close(p[0]);

  action_usr1.sa_sigaction = sig_pere_usr1;
  action_usr1.sa_flags = SA_SIGINFO;
  sigaction(SIGUSR1, &action_usr1, NULL);

  while (1) sleep(1);
}


void sig_fils_usr2(int sig){
  int nombre;
  if (0 > read(p[0], &nombre, sizeof nombre)) perror("read fils");
  printf("%d a lu %d\n", getpid(), nombre);
  nombre = gros_calcul(nombre);
```

```
  printf("%d trouve %d\n", getpid(), nombre);
}

void do_fils () {
  struct sigaction action_usr2;

  close(p[1]);

  action_usr2.sa_handler = sig_fils_usr2;
  sigaction(SIGUSR2, &action_usr2, NULL);

  // printf("Fils %d\n", getpid());
  sleep(2);

  while(1) {
    kill(getppid(), SIGUSR1);
    sleep(2);
  }
}

int main () {
  int i;
  int pere = 1;
  if ( 0 > pipe(p)) perror("No pipe");
  for (i = 0; (i < N) && pere ; i++) {
    pere = fork();
    if (pere < 0) perror("fork");
  }
  if ( pere ) {
    do_pere();
  } else {
    do_fils();
  }
  exit(0);
}
```