Programmation système
18/12/18

François Marès & Doan Nhat Minh

1 Exercice 1

1.1 Partie 1

1.1.1 Prog1

On cherche l'arbre généalogique des processus générés par le programme suivant ; les variables a, b, c et d ont été rajoutées pour qu'il soit plus simple de suivre l'exécution :

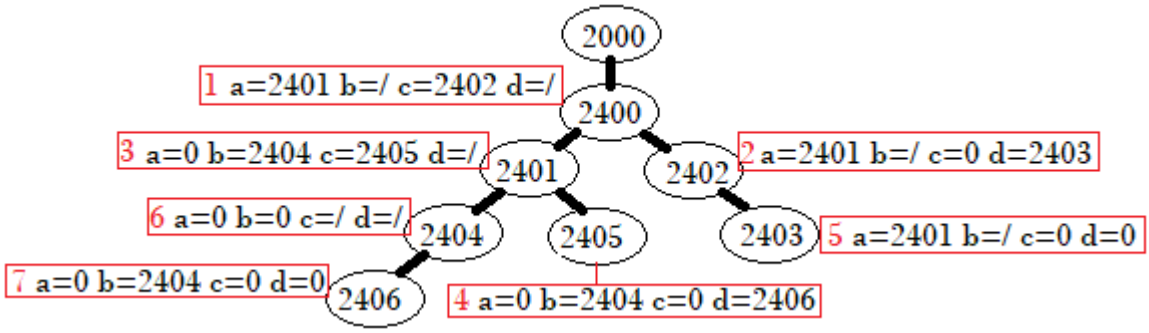
```
#include <unistd.h> /* fork()... */
#include <stdio.h> /* printf... */
#include <stdlib.h> /* EXIT_FAILURE... */

int main (){
  int a=0;
  int b=0;
  int c=0;
  int d=0;
  ( (a=fork()) || (b=fork())) && ( (c=fork()) || (d=fork()));
  printf("Processus : PID=\%d, PPID=\%d, a=\%d, b=\%d, c=\%d, d=\%d\n"
         ,getpid(),getppid(),a,b,c,d);
  sleep(50);
  exit(0);
}
```

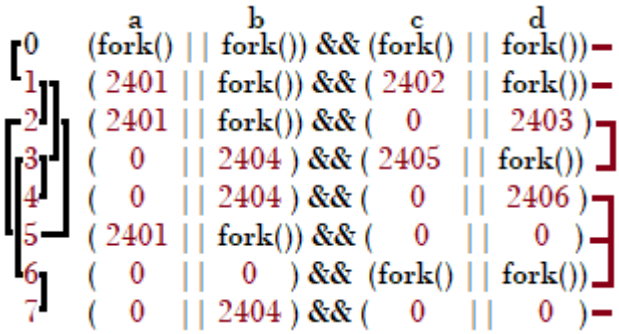
Affichage dans les shells (on visualise l'arborescence dans un deuxième shell avec la commande `watch -n 1 'ps -l --forest -C "bash,ex1"'`) :

```
  PID  PPID CMD                         hannibal@hannibal-VirtualBox:~/Documents$ ./ex1
 1748  1689 bash                        PID=9372, PPID=1699, a=9373, b=1, c=9374, d=1
 1699  1689 bash                        PID=9374, PPID=9372, a=9373, b=1, c=0, d=9375
 9372  1699  \_ ex1                     PID=9373, PPID=9372, a=0, b=9376, c=9377, d=1
 9373  9372      \_ ex1                 PID=9377, PPID=9373, a=0, b=9376, c=0, d=9378
 9376  9373      |   \_ ex1             PID=9375, PPID=9374, a=9373, b=1, c=0, d=0
 9377  9373      |   \_ ex1             PID=9376, PPID=9373, a=0, b=0, c=1, d=1
 9378  9377      |       \_ ex1         PID=9378, PPID=9377, a=0, b=9376, c=0, d=0
 9374  9372      \_ ex1
 9375  9374          \_ ex1
```

Arborescence avec :

- le PID du shell est 2000
- le PID du processus correspondant au parent est 2400
- la numérotation des processus est séquentielle (incrémentation par 1)





Interprétation :

- Pour l'opération x||y, y n'est évalué que si $x = 0$
- Pour l'opération x&&y, y n'est évalué que si $x \neq 0$

Ainsi chaque fois que la valeur de retour d'un fils devient nul ou non-nul, la ligne est réévaluée selon les règles précédentes.

1.1.2 Prog2

```
#include <unistd.h> /* fork()... */
#include <stdio.h> /* printf... */
#include <stdlib.h> /* EXIT_FAILURE... */

int main (){
  int i =0;
  fork ();
  printf("PID=\%d, PPID=\%d i=\%d\n", getpid(), getppid(),i);
  while (i <4) {
    if( getpid () %2==0) {
      fork ();
      printf("PID=\%d, PPID=\%d i=\%d\n", getpid(), getppid(),i);
      }
    i++;
  }
  sleep(50);
  exit(0);
}
```

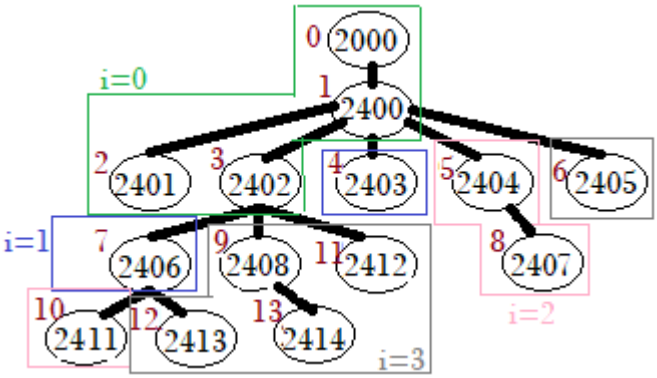
Affichage dans les shells (on visualise l'arborescence dans un deuxième shell avec

la commande `watch -n 1 'ps -l --forest -C "bash,ex12"'`) :

```
  PID  PPID CMD
 1748  1689 bash
 1699  1689 bash
24630  1699  \_ ex12
24631 24630      \_ ex12
24632 24630      \_ ex12
24636 24632      |   \_ ex12
24641 24636      |   |   \_ ex12
24643 24636      |   |   \_ ex12
24638 24632      |   \_ ex12
24644 24638      |   |   \_ ex12
24642 24632      |   \_ ex12
24633 24630      \_ ex12
24634 24630      \_ ex12
24637 24634      |   \_ ex12
24635 24630      \_ ex12
```

```
hannibal@hannibal-VirtualBox:~/Documents$ ./ex12
PID=24630, PPID=1699 i=0
PID=24630, PPID=1699 i=0
PID=24630, PPID=1699 i=1
PID=24630, PPID=1699 i=2
PID=24631, PPID=24630 i=0
PID=24630, PPID=1699 i=3
PID=24632, PPID=24630 i=0
PID=24632, PPID=24630 i=1
PID=24633, PPID=24630 i=1
PID=24634, PPID=24630 i=2
PID=24634, PPID=24630 i=3
PID=24635, PPID=24630 i=3
PID=24636, PPID=24632 i=1
PID=24637, PPID=24634 i=3
PID=24632, PPID=24630 i=2
PID=24632, PPID=24630 i=3
PID=24636, PPID=24632 i=2
PID=24636, PPID=24632 i=3
PID=24638, PPID=24632 i=2
PID=24638, PPID=24632 i=3
PID=24641, PPID=24636 i=2
PID=24642, PPID=24632 i=3
PID=24643, PPID=24636 i=3
PID=24644, PPID=24638 i=3
```

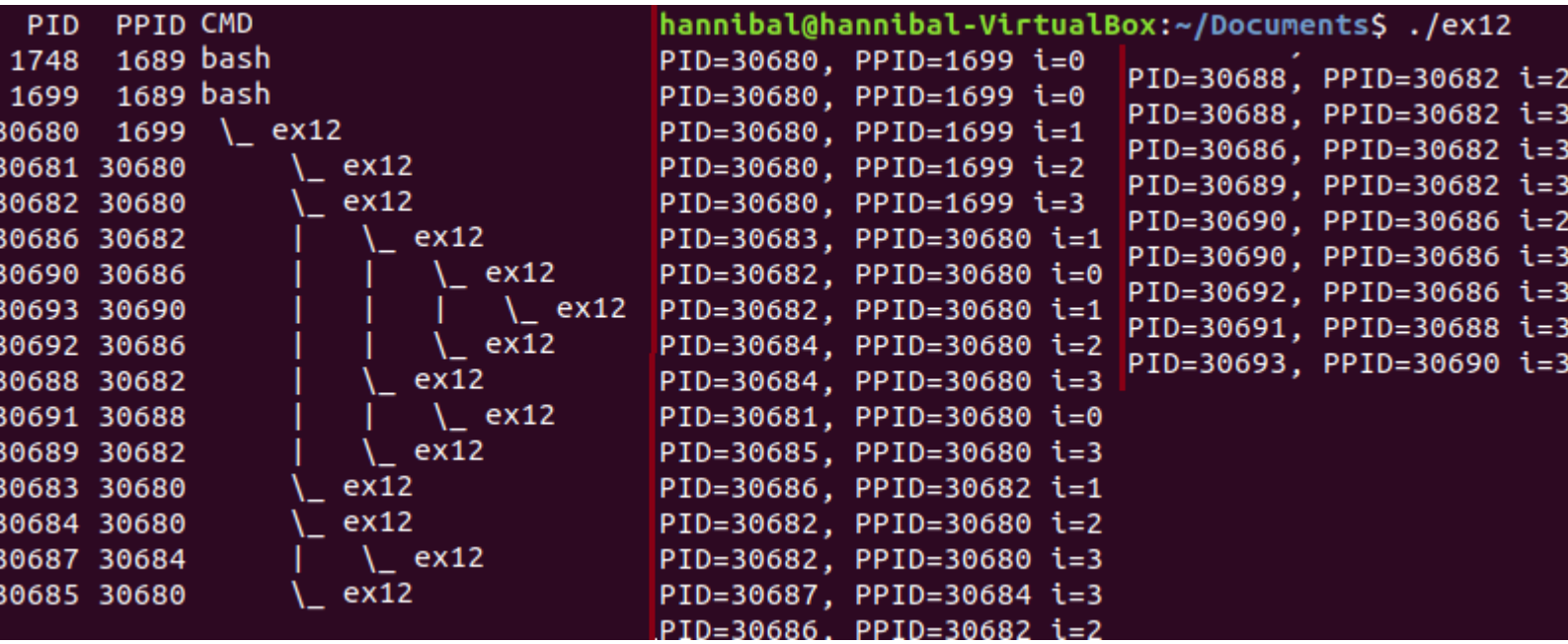
Arborescence avec les mêmes normes :



Le programme s'exécute selon les critères suivants :

- Quand le PID du processus est paire, et que $i < 3$, alors il y a création d'un processus fils.

Autre résultat possible :



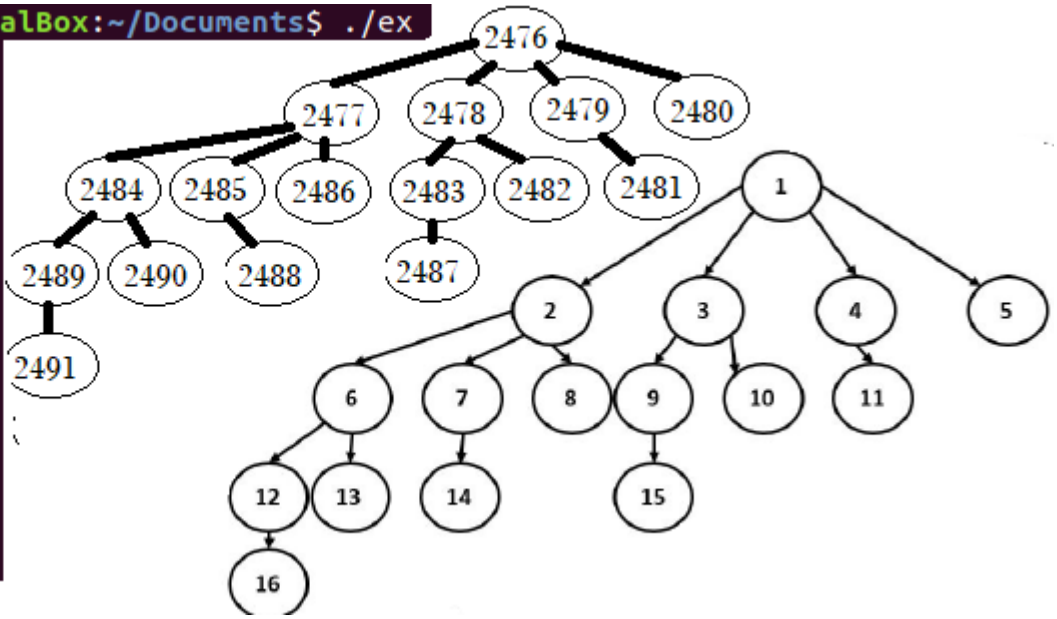
1.2 Partie 2

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

int main()
{
  int i;
  int pid;
  for (int i = 0; i < 4; ++i)
  {
    if ((pid=fork())==-1){
      break;
    }
    if (pid==0){
      printf("PID=\%d PPID=\%d i=\%d\n",getpid(),getppid(),i);
    }
  }
  // Attendre la fin des fils
  while ( wait(NULL) >= 0 );
  return 0;
}
```

- Le wait permet d'attendre le retour de n'importe quel processus fils.

On obtient bien l'arborescence voulue :

```
hannibal@hannibal-VirtualBox:~/Documents$ ./ex
PID=2480 PPID=2476 i=3
PID=2477 PPID=2476 i=0
PID=2478 PPID=2476 i=1
PID=2479 PPID=2476 i=2
PID=2483 PPID=2478 i=3
PID=2484 PPID=2477 i=1
PID=2485 PPID=2477 i=2
PID=2481 PPID=2479 i=3
PID=2482 PPID=2478 i=2
PID=2486 PPID=2477 i=3
PID=2487 PPID=2482 i=3
PID=2488 PPID=2485 i=3
PID=2489 PPID=2484 i=2
PID=2490 PPID=2484 i=3
PID=2491 PPID=2489 i=3
```



2 Exercice 2

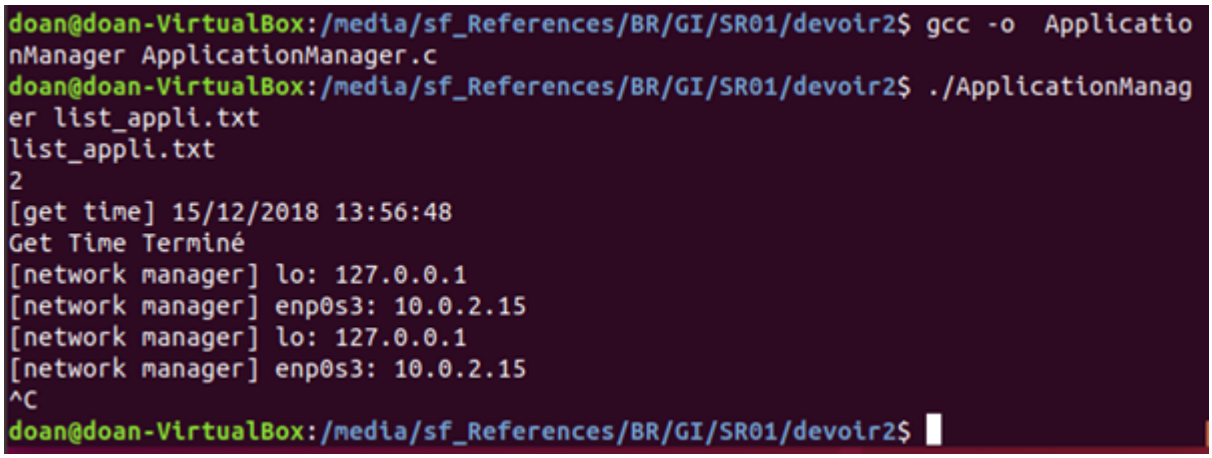
Un gestionnaire d'applications est un programme lancé au démarrage du système d'exploitation. Il se charge de lancer et de gérer un ensemble d'applications nécessaires au bon fonctionnement du système (gestion des cartes réseau, gestion des périphériques...). Dans cet exercice, on va programmer un gestionnaire d'applications personnalisé (Application-Manager).
La liste des applications à lancer est stockée dans le fichier $list_appli.txt.Ondisposedequelquesexemplesd'applications(power manager.c, network_manager.c, get_time.c)$.

```
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=
```

2.1 ApplicationManager.c

On a créé un ensemble de processus fils
tel que chacun soit responsable à l'exécution d'une application grâce à la function :
`int lireProg(int fd, char* appli, char* path, int* argc, char** argv);`
On a déclaré une variable global nbrAppli pout compter le nombre d'applications :

```
for(i=0; i<=nbrAppli; i++){
    while(c[0] == '\n' || c[0] == '\t' || c[0] == ' '){
      if(read(fd, c, 1) <= 0)
        return -1;
    }
    appli[i] = (char*)malloc(MAX*sizeof(char));
    if(lireProg(fd, appli[i], path, &nbrArg, arg) == -1){
      perror("read");
      printf("555\n");
      exit(1);
    }
    pid[i] = fork();                  //creer processus fils
    if(pid[i] < 0){
      perror("fork");
      printf("666\n");
      exit(1);
    }
    if(pid[i] == 0)
      //processus fils
      execv(path, arg);
  }
  close(fd);
```



Pour que le programme ne s'arrête pas 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), on a utilisé une boucle :

```
for(i=0; i<=nbrAppli; i++){
    x = wait(NULL);    //attendre le processus fils
    for(j=0; j<=nbrAppli; j++){
      if(x == pid[j]){
        printf("%s Terminé \n", appli[j]);
        break;
      }
    }
  }
```

2.1.1 SIGUSR1

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

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

void main (int argc , char *argv[]) {
  FILE * fp;
  char c;
  while(1) {
    if(argc != 3) exit(EXIT_FAILURE);
    fp = fopen (argv[1], "r");
    if(fp == NULL) exit(EXIT_FAILURE);
    c = fgetc(fp);
    fclose(fp);
    if(c =='1') {
          printf("[power manager] mise en veille en cours ...\n");
            /* ajoutez vos modification ici */
      fp = fopen (argv[1], "w");
      fputs("0", fp);
      fclose(fp);
      kill(getppid(), SIGUSR1);
      //on n'envoie que SIGUSR1 de power_manager a ApplicationManager
      }
      sleep(atoi(argv[2]));
  }
}
```

3 Exercice 3

3.1 Somme.c

On suppose que deux fichier contiennent chacun une matrice de même taille. Le programme suivant reçoit 3 arguments :

- le chemin du premier fichier
- le chemin du deuxième fichier
- la taille des matrices

Les inclusions nécessaires sont les suivantes :

```
#include <unistd.h> /* fork()... */
#include <stdio.h> /* printf... */
#include <stdlib.h> /* EXIT_FAILURE... */
```

On utilise une fonction de service pour afficher les matrices, qui sont des tableaux de pointeurs sur des tableaux :

```
void print(int **mat,int n);

void print(int **mat,int n){
  int i;
  int j;
  for(i=0; i<n; i++){
    for(j=0; j<n; j++){
      printf("%d ",mat[i][j]);
    }
  printf("\n");
  }
}
```

Dans la fonction main, on commence par verifier le nombre d'arguments passés lors de l'appel :

```
int main(int argc, char *argv[])
{
  if(argc<4){
    return 0;
  }
  int n=atoi(argv[3]);
```

Puis on alloue dynamiquement la mémoire pour les tableaux :

```
  /*Initialisation des matrices (tableau de pointeurs)*/
  int **mat1 = (int **)malloc(n * sizeof(int *));
  int **mat2 = (int **)malloc(n * sizeof(int *));
  int **somme = (int **)malloc(n * sizeof(int *));
  int i;
  for(i=0; i<n; i++){
    mat1[i]= (int *)malloc(n * sizeof(int));
  }
  for(i=0; i<n; i++){
    mat2[i]= (int *)malloc(n * sizeof(int));
  }
  for(i=0; i<n; i++){
    somme[i]= (int *)malloc(n * sizeof(int));
  }
```

Puis on lit les deux fichiers pour initialiser les matrices :

```
/*LECTURE DES FICHIERS*/
  FILE* fichier = NULL;
  fichier = fopen(argv[1], "r");
  if (fichier != NULL){
    for(i=0; i<n; i++){
      fread(mat1[i],sizeof(int),n,fichier);
    }
    fclose(fichier);
  }
  fichier = fopen(argv[2], "r");
  if (fichier != NULL){
    for(i=0; i<n; i++){
      fread(mat2[i],sizeof(int),n,fichier);
    }
    fclose(fichier);
  }
  print(mat1,n);
  print(mat2,n);
```

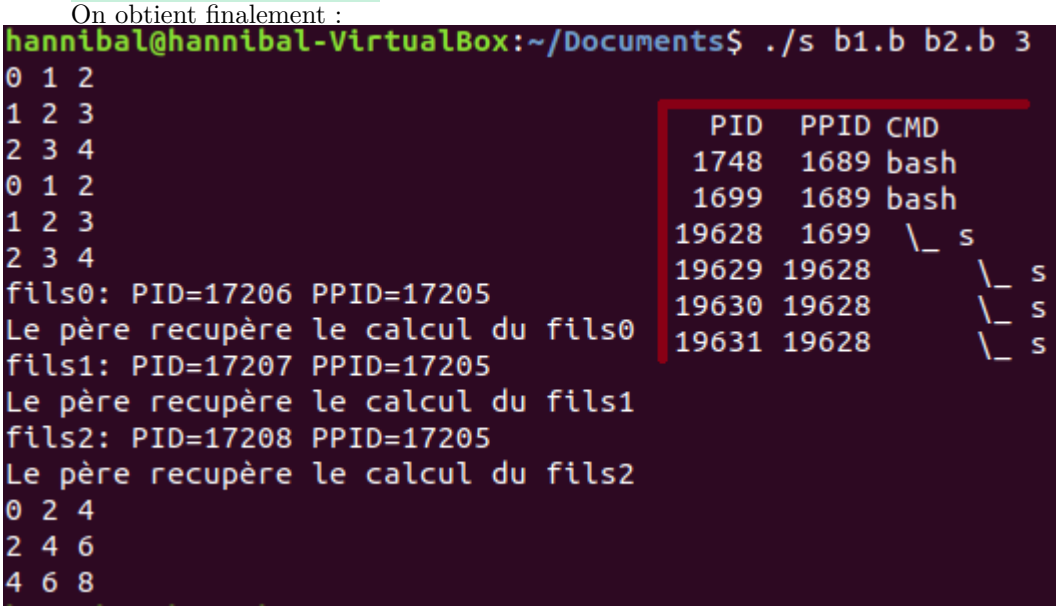
On construit ensuite n processeurs en parallèle qui communiquent avec le processeur père par un tube :

```
  int tube[2];
  int *tabpid=(int*)malloc(n*sizeof(int));
  int j;
  int *ligne=(int*)malloc(n*sizeof(int));
  for (i=0; i < n; i++){
    if (pipe(tube)==-1){
      printf("pipe failed\n");
      exit(EXIT_FAILURE);
    }
        switch(tabpid[i]=fork()){
              case -1: exit(EXIT_FAILURE);
                  break;
              case 0: // Fils
        for(j=0; j<n; j++){
          ligne[j]=mat1[i][j]+mat2[i][j];
        }
        printf("fils%d: PID=%d PPID=%d\n",i,getpid(),getppid());
        close(tube[0]);
                    write(tube[1],&ligne[0],n*sizeof(int));
        exit(0);
                  break;
              default: close(tube[1]);
        read(tube[0],&somme[i][0],n*sizeof(int));
        printf("Le père recupère le calcul du fils%d\n",i);
        close(tube[0]);
        }
  }
  print(somme,n);
```

On fini par libérer la mémoire allouée dynamiquement :

```
free(tabpid);
  free(ligne);
  for(i=0; i<n; i++){
    free(mat1[i]);
  }
  for(i=0; i<n; i++){
    free(mat2[i]);
  }
  for(i=0; i<n; i++){
    free(somme[i]);
  }
  return 0;
}
```

On obtient finalement :



3.1.1 Produit.c

Produit.c reçoit les mêmes paramètres que le programme Somme.c. Il effectue le produit de deux matrices grâce à n processus fils ; chaque processus fils va calculer le produit de la ième ligne de la première matrice avec toutes les colonnes de la deuxième matrice. Ainsi on ne change que les processus fils :

```
                case 0: // Fils
                  for(j=0; j<n; j++){
                     ligne[j]=0;
                     for(k=0;k<n;k++){
                       ligne[j]+=mat1[k][j]+mat2[j][k];
                       }
                  }
                  printf("fils%d: PID=%d PPID=%d\n",i,getpid(),getppid());
                  close(tube[0]);
                  write(tube[1],&ligne[0],n*sizeof(int));
                  exit(0);
```

On obtient finalement :

```
hannibal@hannibal-VirtualBox:~/Documents$ ./p b1.b b2.b 3
0 1 2
1 2 3
2 3 4
0 1 2
1 2 3
2 3 4
fils0: PID=23207 PPID=23206
Le père recupère le calcul du fils0
fils1: PID=23208 PPID=23206
Le père recupère le calcul du fils1
fils2: PID=23209 PPID=23206
Le père recupère le calcul du fils2
6 12 18
6 12 18
6 12 18
```

```
  PID  PPID CMD
 1748  1689 bash
 1699  1689 bash
23206  1699  \_ p
23207 23206      \_ p
23208 23206      \_ p
23209 23206      \_ p
```

3.1.2 ManipMatrice.c

Table des matières