Examen long de système d'exploitation

Département d'informatique – IUT Villetaneuse

Lundi 12 janvier 2009 – 3H

Remarques : tous les documents de cours, td/tp sont autorisés. Le barème est indicatif.

1 Langage de commande bash - 8 pts

A) Écrire un programme Bash qui prend en paramètre quatre entiers a, b, A, B et qui affiche l'ellipse $E(a, b)$, centrée en (a, b), de demi grand axe A et de demi petit axe B. L'ellipse peut être définie par l'équation paramétrique suivante avec $0 \leq \Theta \leq 2\pi$ rad :

$$\begin{cases} x &= a + A \cos\Theta \\ y &= b + B \sin\Theta \end{cases}$$

On obtiendra par exemple le graphique suivant pour l'appel `ellipse 5 7 5 7` :

```
|--------------------------------
|    XXXXXXX
|  XXX     XXX
| XX         XX
|XX           XX
|X             X
|X      .      X
|X             X
|XX           XX
| XX         XX
|  XXX     XXX
|    XXXXXXX
|
|port-cerin:$
|
```

Vous vérifierez que les points calculés sont à l'intérieur de la fenêtre terminal avant de les afficher. Vous utiliserez ou vous vous inspirerez des commandes suivantes :

```
tput cols
     Retourne le nombre de colonnes de la fenêtre courante

tput lines
     Retourne le nombre de lignes de la fenêtre courante

tput cup m n
     Envoie la commande écran qui place le curseur en
     ligne m colonne n. Par exemple, tput cup 0 0 place
     le curseur en position (0,0) (le coin en haut à gauche
     de l'écran, position de curseur appelée 'home'

tput clear
     Envoie la commande d'effacement de l'écran au
     terminal courant.

x=‘echo "3 + 4 * c(2)" | bc -l‘
     Range dans x la valeur 3+ 4*cos(2), L'angle est exprimé en radians.
     Un angle de 1 rad est un angle, qui, ayant son sommet au centre d'un
     cercle, intercepte, sur la circonférence de ce cercle, un arc d'une
     longueur égale à celle du rayon du cercle. Un cercle complet
     représente un angle de 2*pi radians, appelé angle plein.

y=‘echo "3 + 4 * s(2)" | bc -l‘
     Range dans x la valeur 3+ 4*sin(2), L'angle est exprimé en radians.

v=‘python -c "import sys;print round(float(sys.argv[1]))" "12.34"‘
     Assigne à v la valeur arrondie de 12.34 (ici, v=12).

v=‘python -c "import sys;print round(float(sys.argv[1]))" "12.54"‘
     Assigne à v la valeur arrondie de 12.54 (ici, v=13).



#!/bin/bash
#
#echo $1
#echo $2
#echo $3
#echo $4

clear
if test $# = 4
```

```
then
   # calcul de pi
   pi=‘echo ’scale=10;4*a(1)’ | bc -l‘
   angle=0.0
   # nombre max de colonnes et de lignes du terminal
   col=‘tput cols‘
   lig=‘tput lines‘
   # pas d’iteration sur les angles
   increment=‘echo "2*${pi} / 100 "|bc -l‘
   while [ ‘echo "$angle > 2*${pi}"| bc -l‘ -lt 1 ];
   do
     x=‘echo "$1 + $3 * c($angle)"|bc -l‘
     x=‘python -c "import sys;print int(round(float(sys.argv[1])))" $x‘
     y=‘echo "$2 + $4 * s($angle)"|bc -l‘
     y=‘python -c "import sys;print int(round(float(sys.argv[1])))" $y‘
     # on verifie si x et y sont a l’interieur du terminal
     if [ $y -le $col -a $y -ge 0 -a $x -le $lig -a $x -ge 0 ]; then
        tput cup $x $y
        echo -n "X"
#        echo "($col,$lig) $x $y $angle"
     fi
     angle=‘echo "$angle + $increment"|bc -l‘
   done
   # on dessine le centre du cercle
   tput cup $1 $2
   echo -n "."
   # on se déplace en bas de la fenetre
   x=‘tput lines‘
   tput cup $((x-2)) ‘tput cols‘
   echo
else
   echo "Usage - $0 x y A B"
   echo "   x: abscisse, y ordonnee, A: demi grand axe, B: demi petit axe"
   echo "   de l’ellipse"
fi
```

B) En vous souvenant de ce qui a été faite au contrôle court de novembre 2008, quelle est le nom de la figure géométrique produit par l'appel `ellipse 5 7 5 5` ?
C'est un cercle : voir le sujet du contrôle court de novembre 2008. On est dans le cas $A = B$ c'est à dire le cas de l'équation présentée au contrôle court.
C) Écrire une fonction Bash qui teste si un point est à l'intérieur de l'ellipse. Elle affiche Oui ou Non selon que le point est ou n'est pas à l'intérieur. Pour

décider si un point est à l'intérieur de l'ellipse, vous utiliserez l'équation cartésienne de l'ellipse :

$$\frac{(x-a)^2}{A^2} + \frac{(y-b)^2}{B^2} = 1$$

et plus précisément, vous vous intéresserez aux points (x, y) qui rendent la partie gauche de l'égalité plus petite ou égale à 1 (justifiez le en une phrase).

```
#
# Fonction qui controle qu'un point est a l'interieur de l'ellipse
# Paramemetres :
#   ($1,$2) sont les coordonnees du point à tester
#   ($3,$4) sont les coordonnees du centre de l'ellipse
#   $5 est le demi grand axe ; $6 le demi peti axe
#
MyTest()
{
  val=‘echo "(($3 - $1)*($3 - $1))/($5*$5) +
             (($4 - $2)*($4 - $2))/($6*$6) "|bc -l‘
#  echo $val
  val=‘echo "$val <= 1.0"|bc -l‘
  if [ $val -eq 1 ]; then
    echo "$val: ($1,$2) est a l'interieur de l'ellipse"
  else
    echo "$val: ($1,$2) n est pas a l'interieur de l'ellipse"
  fi
}
```

2 Communication locale sous Unix - 8 pts

A) À l'aide des instructions `fork()`, `wait()`, `waitpid()`, `exit()` (au choix) écrire un programme C qui crée 4 processus et qui affichent un des messages suivants : `Titi`, `Toto`, `Tata`, `Tutu`. Le processus père devra terminer après tous ses fils. Voici un exemple de compilation et d'execution :

```
$ gcc -Wall totoititatatutu.c
$ ./a.out
Toto
Tutu
Tata
Titi
Le pere termine
$
```

```c
/* Correction Toto, Tata, Titi, Tutu */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
  int status;
  pid_t wpid, pid1, pid2, pid3;
  /* fork another process */
  pid1 = fork();
  if (pid1 < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid1 == 0) { /* child process */
    printf("Toto\n");
    exit(0);
  }

  /* fork another process */
  pid2 = fork();
  if (pid2 < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid2 == 0) { /* child process */
    sleep(1);
    printf("Titi\n");
    exit(0);
  }

  /* fork another process */
  pid3 = fork();
  if (pid3 < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    exit(-1);
  }
  else if (pid3 == 0) { /* child process */
    printf("Tata\n");
    exit(0);
  }
```

```
  else { /* parent process */
    printf("Tutu\n");
    /* parent will wait for the children to complete */
    /* On synchronise les fils */
    wpid = waitpid(pid2, &status, WUNTRACED);
    wpid = waitpid(pid3, &status, WUNTRACED);
    wpid = waitpid(pid1, &status, WUNTRACED);
    /* Le pere peut terminer */
    printf("Le pere termine\n");
    exit(0);
  }
}
```

B) J'ai envoyé par courrier éléctronique une proposition de correction d'un sujet de TP qui consistait à mettre en place 2 écrivains et 2 consommateurs. Les 2 écrivains partageaient le même tube pour communiquer si bien que l'entrée du tube était protégée par un sémaphore (afin de garantir l'accès exclusif au tube). Le code était le suivant :

```
/*
 * Ce programme cree deux processus qui generent aleatoirement des
 * entiers qui sont passés à 2 tubes  (selon le critere pair/impair)
 * Il y a un acces exclusif a faire sur les tubes car les processus
 * s'exécutent concurremment. Dans le code qui suit, il n'y a une
 * gestion de l'acces exclusif que sur le tube desc1
 *
 * Faire un man de sem_post, sem_wait et sem_init pour comprendre
 * ce qui se passe dans les fonctions P() et V() qui implemente
 * la notion de semaphore. A ce sujet, voir Wikipedia sur
 * http://fr.wikipedia.org/wiki/Sémaphore_(informatique)
 *
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
#include <stdlib.h>
#include <semaphore.h>
/*
 * On reprend la structure d'un exemple precedent
 * mais il n'y a que le semaphore shared.full qui est utilise
*/
static struct
{
```

```
  int buf;          /* shared var */
   sem_t full;          /* sems */
   sem_t empty;
} shared;
/*
 * Operation pour acquerir la section critique
*/
void
P (sem_t * sem)
{
  if (sem_wait (sem))
     printf ("P\n");
}

/*
 * Operation pour relacher la section critique
*/
void
V (sem_t * sem)
{
  if (sem_post (sem))
     printf ("V\n");
}

int
main (void)
{
  int i, pid;
  int c;
  int desc1 = open ("tub1", O_WRONLY);
  int desc2 = open ("tub2", O_WRONLY);
  if (desc1 == -1 && desc2 == -1)
  {
      fprintf (stderr, "Erreur : tube \n");
      exit (1);
  }

  /* initialize the semaphore */
  /* sem_init(&shared.empty, 0, 1); */
  sem_init (&shared.full, 0, 1);
  pid = fork ();
  if (pid == -1)
    perror ("je sors\n");
  else
  {
   if (pid == 0)
   {
```

```
     /* initialisation du germe du generateur aleatoire */
     srand (13);
     for (i = 0; i < 5; i++)
       {
         c = 1 + (int) (100.0 * (rand () / (RAND_MAX + 1.0)));
         printf ("FILS a genere %d\n", c);
         if (c % 2 == 0)
         {
           P (&shared.full);
           write (desc1, &c, sizeof (int));
           V (&shared.full);
         }
         else
           write (desc2, &c, sizeof (int));
       }
       exit(1); /* Le fils sort */
   }
   else
   {
     /* initialisation du germe du generateur aleatoire */
     srand (17);
     for (i = 0; i < 5; i++)
     {
         c = 1 + (int) (100.0 * (rand () / (RAND_MAX + 1.0)));
         printf ("Pere a genere %d\n", c);
         if (c % 2 == 0)
         {
           P (&shared.full);
           write (desc1, &c, sizeof (int));
           V (&shared.full);
         }
         else
           write (desc2, &c, sizeof (int));
    }}}
    /* Le pere attend le fils avant de fermer les tubes */
    wait (0);
    /* comme le fils est sorti par exit(), seul le pere ferme les tubes */
    close (desc1);
    close (desc2);
    exit (0);
}



/* Lecteur 1 */
```

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int
main (void)
{
  int desc;
  int somme = 0;
  int i;
  mkfifo ("tub1", 0644);
  desc = open ("tub1", O_RDONLY);
  while (read (desc, &i, sizeof (int)))
  {
      somme = somme + i;
      printf (" somme pairs = %d (+%d)\n", somme, i);
  }
  close (desc);
  remove ("tub1");
}

/* lecteur 2 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int
main (void)
{
  int desc;
  int somme = 0;
  int i;
  mkfifo ("tub2", 0644);
  desc = open ("tub2", O_RDONLY);
  while (read (desc, &i, sizeof (int)))
  {
      somme = somme + i;
      printf (" somme impairs = %d (+%d)\n", somme, i);
  }
  close (desc);
  remove ("tub2");
}
```

B.1) Dans ce code il y a 3 exécutables (produits par la compilation des 3 fichiers sources C). On vous demande de réécrire les codes afin de ne garder qu'un seul fichier source. Pour cela, vous allez créer des procesus qui vont jouer le rôle du premier producteur, du deuxième producteur, du premier consommateur et du deuxième consommateur. Commencez par faire des explications pour exposer les difficultés et comment vous vous y prenez.

```
/*
 * Deux ecrivains et deux lecteurs sont activés dans ce code.
 * Attention : il n'y a pas de semaphore sur une des deux écritures
 * (a faire à titre d'exercice)
 *
 * Il y a automatiquement synchronisation des processus qui
 * ouvrent en mode bloquant un tube nommé.
 * L'opération d'ouverture sur un tube nommé est bloquante en lecture.
 * Le processus attend qu'un autre processus ouvre la fifo en écriture.
 * L'ouverture en écriture est aussi bloquante, avec attente qu'un
 * autre processus ouvre la fifo en lecture. L'ouverture bloquante
 * se termine de façon synchrone pour les deux processus.
 *
 * Ainsi un unique processus ne peut ouvrir à la fois en lecture
 * et écriture un tube nommé.
 *
 * A compiler par : gcc -O4 -Wall 2ecri2lecFORK.c -lpthread
 *
 * Note : de la memoire partagee est aussi utilisee ici
 */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdlib.h>
#include <semaphore.h>

/*
 * On reprend la structure d'un exemple precedent mais il n'y a que le
 * semaphore shared.full qui est utilise
 */
static struct {
     int             buf;     /* shared var */
     sem_t           full;     /* sems */
```

```
     sem_t           empty;
}               shared;

/*
 * Operation pour acquerir la section critique
*/
void
P(sem_t * sem)
{
     if (sem_wait(sem))
          printf("P\n");
}


/*
 * Operation pour relacher la section critique
*/
void
V(sem_t * sem)
{
     if (sem_post(sem))
          printf("V\n");
}

int
main(int argc, char *argv[])
{
     int     status, c, somme, i, fd;
     pid_t   wpid, pid1, pid2, pid3;

     struct region {          /* Defines "structure" of shared memory */
          int  descR1, descR2, descW1, descW2;

     };
     struct region  *rptr;

     fd = open("/tmp/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

     /* utilise toute la zone de mem */
     if (ftruncate(fd, sizeof(struct region)) == -1) {
     }
     /* Map shared memory object */
     rptr =
          mmap(NULL, sizeof(struct region), PROT_READ | PROT_WRITE, MAP_SHARED,
               fd, 0);
     if (rptr == MAP_FAILED) {
     }
```

```
/* creation des tubes et ouvertures */
if (mkfifo("tub1", 0644) != 0) {
     fprintf(stderr, "Erreur : mkfifo tub1\n");
     exit(1);
}
if (mkfifo("tub2", 0644) != 0) {
     fprintf(stderr, "Erreur : mkfifo tub2\n");
     exit(1);
}

/* initialize the semaphore */
/* sem_init(&shared.empty, 0, 1); */
sem_init(&shared.full, 0, 1);

/* fork another process */
pid1 = fork();
if (pid1 < 0) {          /* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
} else if (pid1 == 0) {     /* child process */
     somme = 0;
     rptr->descR1 = open("tub1", O_RDONLY);
     if (rptr->descR1 == -1) {
          fprintf(stderr, "Erreur : tube descR1\n");
          exit(1);
     }
     while (read(rptr->descR1, &i, sizeof(int))) {
          somme = somme + i;
          printf(" somme pairs = %d (+%d)\n", somme, i);
     }
     printf("fils 1 termine\n");
     close(rptr->descR1);
     exit(0);
}
/* fork another process */
pid2 = fork();
if (pid2 < 0) {          /* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
} else if (pid2 == 0) {     /* child process */
     somme = 0;
     rptr->descR2 = open("tub2", O_RDONLY);
     if (rptr->descR1 == -1) {
          fprintf(stderr, "Erreur : tube descR2\n");
          exit(1);
     }
     while (read(rptr->descR2, &i, sizeof(int))) {
```

```
            somme = somme + i;
            printf(" somme impairs = %d (+%d)\n", somme, i);
        }

        printf("fils 2 termine\n");
        close(rptr->descR2);
        exit(0);
}
/* fork another process */
pid3 = fork();
if (pid3 < 0) {          /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
} else if (pid3 == 0) {     /* child process */
        rptr->descW1 = open("tub1", O_WRONLY);
        if (rptr->descW1 == -1) {
            fprintf(stderr, "Erreur : tube descW1\n");
            exit(1);
        }
        rptr->descW2 = open("tub2", O_WRONLY);
        if (rptr->descW2 == -1) {
            fprintf(stderr, "Erreur : tube descW2\n");
            exit(1);
        }
        srand(5);
        for (i = 0; i < 5; i++) {
            c = 1 + (int) (100.0 * (rand() / (RAND_MAX + 1.0)));
            printf("FILS a genere %d\n", c);
            if (c % 2 == 0) {
                P(&shared.full);
                write(rptr->descW1, &c, sizeof(int));
                V(&shared.full);
            } else
                write(rptr->descW2, &c, sizeof(int));

        }
        printf("fils 3 termine\n");
        close(rptr->descW1);
        close(rptr->descW2);
        exit(1);     /* Le fils sort */
} {              /* parent process */

        rptr->descW1 = open("tub1", O_WRONLY);
        if (rptr->descW1 == -1) {
            fprintf(stderr, "Erreur : tube descW1\n");
            exit(1);
        }
```

```
            rptr->descW2 = open("tub2", O_WRONLY);
            if (rptr->descW2 == -1) {
                 fprintf(stderr, "Erreur : tube descW2\n");
                 exit(1);
            }
            srand(17);
            for (i = 0; i < 5; i++) {
                 c = 1 + (int) (100.0 * (rand() / (RAND_MAX + 1.0)));
                 printf("PERE a genere %d\n", c);
                 if (c % 2 == 0) {
                      P(&shared.full);
                      write(rptr->descW1, &c, sizeof(int));
                      V(&shared.full);
                 } else
                      write(rptr->descW2, &c, sizeof(int));
            }

            close(rptr->descW1);
            close(rptr->descW2);

            /* parent will wait for the children to complete */
            /* On synchronise les fils */
            wpid = waitpid(pid3, &status, WUNTRACED);
            wpid = waitpid(pid1, &status, WUNTRACED);
            wpid = waitpid(pid2, &status, WUNTRACED);

            /* Le pere peut terminer */
            printf("Le pere termine\n");
            /* on detruit les tubes */
            remove("tub1");
            remove("tub2");

            /* on ferme la zone de memoire partagee */
            munmap(rptr, sizeof(struct region));
            /*
                   shm_unlink("/tmp/myregion");
            */
            close(fd);

            exit(0);
       }

}
```

B.2) Vous modifierez ensuite votre code pour ne communiquer qu'avec un seul tube et de la façon suivante. Deux consommateurs écrivent concurremment sur le tube sans tenir compte du fait que l'entier généré soit pair ou impair. Deux

lecteurs lisent concuremment sur le tube et de manière alternative : le premier consommateur lit, puis ce sera au tour du deuxième, puis du premier, puis du deuxième, puis du premier...
Vous commencerez par expliquer ce que vous êtes amenés à faire avant de le coder.

```
/*
 * Deux ecrivains et deux lecteurs sont activés dans ce code.
 * Un seul tube de communication est mis en jeu.
 * Les lecteurs se passent tour a tour les droits de lecture
 * sur le tube via un semaphore. Dans la section critique, un lecteur
 * teste une variable partagee et selon sa valeur decide de lire
 * sur le tube ou de ne rien faire. Le semaphore permet ici
 * d'assurer à la fois l'acces exclusif au tube et de controler
 * le lecteur (ici en choisir un parmi 2) qui accede.
 *
 *
 * Il y a automatiquement synchronisation des processus qui
 * ouvrent en mode bloquant un tube nommé.
 * L'opération d'ouverture sur un tube nommé est bloquante en lecture.
 * Le processus attend qu'un autre processus ouvre la fifo en écriture.
 * L'ouverture en écriture est aussi bloquante, avec attente qu'un
 * autre processus ouvre la fifo en lecture. L'ouverture bloquante
 * se termine de façon synchrone pour les deux processus.
 *
 * Ainsi un unique processus ne peut ouvrir à la fois en lecture
 * et écriture un tube nommé.
 *
 * A compiler par : gcc -O4 -Wall 2ecri2lecFORK.c -lpthread
 *
 * Note : de la memoire partagee est aussi utilisee ici
 */

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdlib.h>
#include <semaphore.h>

/*
 * On reprend la structure d'un exemple precedent mais il n'y a que le
 * semaphore shared.full qui est utilise
```

```
 */
static struct {
     int             buf;     /* shared var */
     sem_t           full;     /* sems */
     sem_t           QuelLecteur;
}               shared;

/*
 * Operation pour acquerir la section critique
*/
void
P(sem_t * sem)
{
     if (sem_wait(sem))
          printf("P\n");
}


/*
 * Operation pour relacher la section critique
*/
void
V(sem_t * sem)
{
     if (sem_post(sem))
          printf("V\n");
}

int
main(int argc, char *argv[])
{
  int     status, c, somme, i, fd, fd1;
     pid_t   wpid, pid1, pid2, pid3;

     struct region {          /* Defines "structure" of shared memory */
       int  descR1, descW1, descR2, descW2;
       /* variable partagee permetant de passer la main */
       /* tour a tour aux differents processus */
       int Myturn;
     };
     struct region  *rptr;

     fd = open("/tmp/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);

     /* utilise toute la zone de mem */
     if (ftruncate(fd, sizeof(struct region)) == -1) {
     }
```

```
/* Map shared memory object */
rptr =
     mmap(NULL, sizeof(struct region), PROT_READ | PROT_WRITE, MAP_SHARED,
          fd, 0);
if (rptr == MAP_FAILED) {
}
/* creation des tubes et ouvertures */
if (mkfifo("tub1", 0644) != 0) {
     fprintf(stderr, "Erreur : mkfifo tub1\n");
     exit(1);
}

/* initialize the semaphores */
sem_init(&shared.full, 0, 1);
sem_init(&shared.QuelLecteur, 0, 1);

/* initialise la variable partagee a 1 */
rptr->Myturn=1;

/* fork another process */
pid1 = fork();
if (pid1 < 0) {          /* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
} else if (pid1 == 0) {     /* child process */
     somme = 0;
     rptr->descR1 = open("tub1", O_RDONLY);
     if (rptr->descR1 == -1) {
          fprintf(stderr, "Erreur : tube descR1\n");
          exit(1);
     }
     fd1=1;i=0;
     while (fd1>0) {
          P(&shared.QuelLecteur);
  if(rptr->Myturn == 0) {
            fd1=read(rptr->descR1, &i, sizeof(int));
            /* on passe la main ou suivant */
            rptr->Myturn++;
            somme = somme + i;
            printf(" somme fils 1 = %d (+%d)\n", somme, i);
  }
          V(&shared.QuelLecteur);
     }
     printf("fils 1 termine\n");
     close(rptr->descR1);
     exit(0);
}
```

```
/* fork another process */
pid2 = fork();
if (pid2 < 0) {          /* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
} else if (pid2 == 0) {     /* child process */
     somme = 0;
     rptr->descR2 = open("tub1", O_RDONLY);
     if (rptr->descR2 == -1) {
          fprintf(stderr, "Erreur : tube descR2\n");
          exit(1);
     }
     fd1=1;i=0;
     while (fd1>0) {
          P(&shared.QuelLecteur);
  if(rptr->Myturn == 1) {
            fd1=read(rptr->descR2, &i, sizeof(int));
            /* on passe la main ou suivant */
            rptr->Myturn--;
            somme = somme + i;
            printf(" somme fils 2 = %d (+%d)\n", somme, i);
  }
          V(&shared.QuelLecteur);
     }

     printf("fils 2 termine\n");
     close(rptr->descR2);
     exit(0);
}
/* fork another process */
pid3 = fork();
if (pid3 < 0) {          /* error occurred */
     fprintf(stderr, "Fork Failed");
     exit(-1);
} else if (pid3 == 0) {     /* child process */
     rptr->descW1 = open("tub1", O_WRONLY);
     if (rptr->descW1 == -1) {
          fprintf(stderr, "Erreur : tube descW1\n");
          exit(1);
     }
     srand(5);
     for (i = 0; i < 5; i++) {
          P(&shared.full);
          c = 1 + (int) (100.0 * (rand() / (RAND_MAX + 1.0)));
          printf("FILS a genere %d\n", c);
          write(rptr->descW1, &c, sizeof(int));
          V(&shared.full);
```

```
        }
        printf("fils 3 termine\n");
        close(rptr->descW1);
        exit(1);      /* Le fils sort */
    } {               /* parent process */

        rptr->descW2 = open("tub1", O_WRONLY);
        if (rptr->descW2 == -1) {
            fprintf(stderr, "Erreur : tube descW1\n");
            exit(1);
        }
        srand(17);
        for (i = 0; i < 5; i++) {
            P(&shared.full);
            c = 1 + (int) (100.0 * (rand() / (RAND_MAX + 1.0)));
            printf("PERE a genere %d\n", c);
            write(rptr->descW2, &c, sizeof(int));
            V(&shared.full);
        }
        close(rptr->descW2);

        /* parent will wait for the children to complete */
        wpid = waitpid(pid3, &status, WUNTRACED);
        wpid = waitpid(pid1, &status, WUNTRACED);
        wpid = waitpid(pid2, &status, WUNTRACED);

        /* Le pere peut terminer */
        printf("Le pere termine\n");
        /* on detruit les tubes */
        remove("tub1");
        remove("tub2");

        /* on ferme la zone de memoire partagee */
        munmap(rptr, sizeof(struct region));
        /*
               shm_unlink("/tmp/myregion");
        */
        close(fd);

        exit(0);
    }

}
```

3 Ordonnancement – 4pts

A) Soient les données d'ordonnancement suivantes :

Processus	Date d'arrivée	Temps de traitement
A	0	2
B	6	5
C	2	2
D	1	4
E	5	3

A) Quel est le déroulement de l'exécution des processus pour l'algorithme SRT ? (on préempte à chaque unité de temps). Même question pour l'algorithme du tourniquet (quantum de 1). On dispose que d'un seul cœur.

B) Même question en supposant que vous disposez maintenant de deux cœurs sur votre processeur. Est-ce que les temps d'exécution sont meilleurs ?