SY40 TP7 /yann derré

a) Sémaphores

a) prg1.c

Le programme 1 a été modifié pour prendre en charge le programme sémaphore.

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include "semaphore.c.c"

typedef struct {
    int semId;
} Semaphore;

Semaphore plouaret;
Semaphore lannion;
```

J'importe la librairie sémaphore, et initialise mon sem plouaret et lannion

```c
/* creation du processus TGV */
void TGV(int i)
{
if (! fork())
{
    message(i, "depart Paris");
    attente(3);
    message(i, "arrivee Plouaret");
    attente(3);
    V(plouaret.semId);
    message(i, "depart Plouaret");
    attente(10);
    message(i, "arrivee Brest");
    attente(3);
    message(i, "arret");
    exit(0);
}
}

/* creation du processus autorail */
void autorail(int i)
{
if (! fork())
{
    message(i, "attente train");
    attente(3);
    P(plouaret.semId);
    message(i, "depart Plouaret");
    attente(3);
    message(i, "arrivee Lannion");
    attente(3);
    V(lannion.semId);
    message(i, "arret");
    exit(0);
}
}
```

Il suffit de placer les P et V de la même manière en précisant cette fois le semId de l'objet.

```c
int main()
{
int i;
plouaret.semId=0;
lannion.semId=1;
printf("%10s%20s%20s\n\n", "TGV", "AUTORAIL", "TAXI");
initSem(2, "prg1.c", NULL);
TGV(0);
autorail(1);
taxi(2);
for (i=1; i<=3; i++) wait(0);
libereSem();
exit(0);
}
```

```
(base) yannderre@MacBook-Pro-2 [20:21:39] [~/Downloads/TP7/sources/ou
-> % ./"prg1.c"
       TGV            AUTORAIL                TAXI

semaphore 0 initialise a 0
semaphore 1 initialise a 0
depart Paris
                    attente train
                                          attente autorail
arrivee Plouaret
depart Plouaret
                    depart Plouaret
                    arrivee Lannion
```

L'output de la fonction, foncitonne comme attendu

b) prg2.c

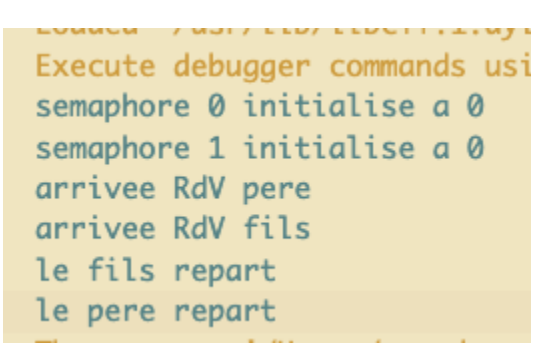
On réutilise la structure du prg1.c

```c
void childProcess()
{
    printf("arrivee RdV fils\n");
    V(semaphores[0].semId); // Signal that child has arrived
    P(semaphores[1].semId); // Wait for parent to arrive
    printf("le fils repart\n");
    exit(0);
}

void parentProcess()
{
    printf("arrivee RdV pere\n");
    V(semaphores[1].semId); // Signal that parent has arrived
    P(semaphores[0].semId); // Wait for child to arrive
    printf("le pere repart\n");
    exit(0);
}
```

```c
int main()
{
    initSem(2, "prg2.c", NULL); // Initialize semaphores
    semaphores[0].semId = 0; // Set the semaphore IDs
    semaphores[1].semId = 1;
    if (fork() == 0)
    {
        // Child process
```

La fonction main initialise les id sémaphores pour pouvoir les utiliser dans les fonction child et parentProcess()



L'output de la fonction

D'autres modifications ont dû être faites. Les programmes ne peuvent s'exécuter qu'en debugger, le programme sémaphore produisant un comportement inconsistant à chaque lancement normal de la fonction SemInit().

Les imports ont également dû être modifiés, les programmes étaient normés "*.c.c".

b) Problème des philosophes

```c
#define NUM_PHILOSOPHERS 5

typedef struct {
    int id;
    int left_fork;
    int right_fork;
} Philosopher;

typedef struct
{
    int semId;
} Semaphore;
```

La structure pour les couverts du philosophe

```c
int main() {
    srand(time(NULL));

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        initSem(1, "semaphore.c.c", NULL);
    }


    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        philosophers[i].id = i;
        philosophers[i].left_fork = i;
        philosophers[i].right_fork = (i + 1) % NUM_PHILOSOPHERS;

        if (fork() == 0) {
            philosopher_process(i);
        }
    }


    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        wait(NULL);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        libereSem();
    }

    return 0;
}
```

On initialise les 5 sémaphores philosophes. Le couvert de droite sera forcément directement après celui de gauche (grâce au modulo 5, i.e. le 5e philosophe prendra le couvert gauche 4 et droit 0. Pour chaque philosophe, on fork la fonction qui enchaîne penser, prendre les couverts, manger et reposer les couverts. On attend la fin des process fils, et on libère les sémaphores

```
void philosopher_process(int philosopher_id) {
    while (1) {
        think(philosopher_id);
        take_forks(philosopher_id);
        eat(philosopher_id);
        put_forks(philosopher_id);
    }
}
```

Chaque philosophe agit dans cet ordre

```
void think(int philosopher_id) {
    printf("Philosopher %d is thinking.\n", philosopher_id);
    sleep(rand() % 2);
}

void eat(int philosopher_id) {
    printf("Philosopher %d is eating.\n", philosopher_id);
    sleep(rand() % 2);
}

void take_forks(int philosopher_id) {
    V(forks[philosopher_id].semId);
    V(forks[(philosopher_id + 1) % NUM_PHILOSOPHERS].semId);
        printf("Philosopher %d takes the fork.\n", philosopher_id);

}

void put_forks(int philosopher_id) {
    P(forks[philosopher_id].semId);
    P(forks[(philosopher_id + 1) % NUM_PHILOSOPHERS].semId);
    printf("Philosopher %d puts the fork back.\n", philosopher_id);

}
```

Chaque philosophe va incrémenter puis décrémenter le process pour assurer que seuls deux philosophes à l'id "éloignés de 2" mangent ensemble.

```
Philosopher 0 puts the fork back.
Philosopher 0 is thinking.
Philosopher 1 takes the fork.
Philosopher 1 is eating.
Philosopher 3 takes the fork.
Philosopher 3 is eating.
Philosopher 1 puts the fork back.
Philosopher 3 puts the fork back.
Philosopher 3 is thinking.
```

Le résultat de la fonction, comportement attendu