TP $n+2$: Fils d'exécution et élections

Exercice 1 : Ensembles de Julia par itération inverse.

On peut approcher l'ensemble de Julia d'une fonction complexe en itérant son inverse un nombre suffisant de fois. Étudions la (classique) fonction, avec $c \in \mathbb{C}$,

$$\begin{aligned} Q_c : \mathbb{C} &\to \mathbb{C} \\ z &\mapsto z^2 + c \end{aligned}$$

On définit l'orbite à rebours comme $\{f_c^{-n}(z_0)\,; n \in \mathbb{N}\}$. Pour la calculer, on note que si $z^2 + c = w$, alors $z = \rho \exp(i\theta)$ avec $\rho = \sqrt{|w-c|}$ et $\theta = \frac{\vartheta}{2} + \delta\pi$ avec $\delta \in \{0\,, 1\}$ et

$$\vartheta = \arctan(\Im(w-c)/\Re(w-c)) + \begin{cases} 0 & \text{si } \Re(w-c) > 0 \\ \pi & \text{sinon} \end{cases}$$

Le point initial w_0 n'a pas d'importance. On se propose de créer un fil d'exécution par orbite, chaque orbite ayant un point initial différent.

On notera les points ou bien sur la sortie standard, ou bien dans un fichier, chaque ligne étant de la forme `x y`. Si les points sont notés dans un fichier nommé `julia.dat`, on peut tracer avec la commande `gnuplot julia.p` en supposant l'existence du script figure 1. On n'oubliera pas d'importer `math.h` et

```
set terminal png size 500,500
set output 'julia.png'
set title 'Julia set'
plot 'julia.dat'
```

FIG. 1: Script gnuplot pour tracer l'ensemble de Julia à partir d'un fichier de données `julia.dat`

`complex.h` et de compiler avec `-lm` en *fin de ligne de commande*. La commande suivant devrait convenir, `gcc julia.c -lpthread -lm`.

On pourra essayer de tracer l'ensemble de Julia avec les paramètres $c \in \{-1\,, -0\,.4 - 0\,.6i\,, -1\,.5\,, -i\,, -0\,.8 + 0\,.4i\,, 0\,.5\,, 3\,, 1+i\,, 2\}$

Solution :

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>
#include <complex.h>

#define THREADS 6
```

```c
int _orbit_len = 400;
double complex _c = -0.4 - 0.6 * I;

double complex inverse(double complex w) {
    double rho  = sqrt(cabs(w - _c));
    double rwmc = creal(w - _c);
    double vtheta = atan(cimag(w - _c) / rwmc);
    if (rwmc <= 0) {
        vtheta = vtheta + M_PI;
    }
    double theta = vtheta / 2;
    if (random() % 2) { theta = theta + M_PI; }
    return rho * cexp(I * theta);
}

void orbit(complex double w0) {
    double complex w = w0;
    for (int i = 0; i < _orbit_len; i++) {
        printf("%lf %lf\n", creal(w), cimag(w));
        w = inverse(w);
    }
    pthread_exit(NULL);
}

double randf(double min, double max) {
    return min + ((float) rand() / (float) (RAND_MAX)) * (max - min);
}

int main() {
    pthread_t orbit_th[THREADS];
    for (int i = 0; i < THREADS; i++) {
        int w = randf(-1, 1) + I * randf(-1, 1);
        pthread_create(orbit_th + i, NULL, (void *(*)(void*)) orbit, &w);
    }
    for (int i = 0; i < THREADS; i++)
        pthread_join(orbit_th[i], NULL);
    exit(0);
}
```

Exercice 2 : Serveur de dates: exclusion mutuelle simple.

On souhaite mettre en œuvre une architecture client serveur (sans pour autant utiliser l'API socket, on se contentera d'avoir des fils d'exécution client et un fil serveur) dans laquelle le serveur est chargé de fournir la date chaque fois que le client la demande.

Le serveur tourne en boucle infinie. Il attend une requête de la part des clients. Une fois la requête reçue, il émet une chaîne de caractères contenant la date et l'heure à disposition du client. À ce moment là, le serveur est prêt à répondre à une autre requête.

La tâche cliente exécutera 50 fois une fonction réalisant :

- émission d'une requête au serveur,
- récupération de la date envoyée au serveur,

- affichage numéroté de la date.

(a) Quelle information est partagée entre les threads ? Quel moyen de communication sera utilisé ?

(b) Quel moyen de synchronisation sera utilisé ?

(c) Implémenter la solution avec plus d'un client.

Exercice 3 : Élection d'un meneur.

Un problème important dans la programmation concurrente est de choisir un *leader*, e.g. pour l'organisation et la coordination d'un réseau. On part avec un nombre n d'ordinateurs ou de processus (on parlera des *nœuds*) à priori identiques, à l'exception d'un identifiant unique. En suivant un même protocole, tous les nœuds vont s'accorder sur le choix d'un leader.

Il existe une grande variété de protocoles menant à ce résultat. Nous allons dans ce TP en étudier et en réaliser un : le protocole Dolev, Klawe, and Rodeh qui se distingue par le faible nombre de messages échangé pour l'élection : $\mathcal{O}(n \log n)$. Par comparaison, les approches simples ont tendance à utiliser $\mathcal{O}(n^2)$ messages. Le protocole a été presenté en cours et les diapos explicatives sont aussi disponibles dans le répertoire du TD.

Le protocole part du principe que les nœuds sont organiseés en anneau, chaque nœud envoie des messages à son voisin de droite. Le répertoire du TP contient un squelette qui créera n processus (les nœuds) pour n donné ; ces processus seront déjà équipés de pipes pour communiquer. Votre tâche est de compléter le programme en réalisant le protocole :

- Au départ, tous les nœuds sont *actifs* et possèdent un identifiant unique.
- Un nœud actif attend une petite période aléatoire (fonction `delay()`), puis envoie son identifiant à son voisin de droite. Ensuite, il attend les identifiants de ses *deux* plus proches voisins actifs à sa gauche. Il décidera ensuite s'il reste actif, devient passif, ou se déclare leader. Les conditions seront précisées dans la présentation. S'il reste actif, il répète le comportement présenté ci-dessus.
- Un nœud *passif* transmet simplement tous les messages reçus de gauche vers son voisin de droite. En plus, si le message déclare un leader, il affiche un message correspondant à l'écran.
- Il y a trois types de messages échangés par les nœuds, tous dans le format (*type*,*identifiant*).
  - "voisin" (v) : pour envoyer son identifiant vers le voisin de droite.
  - "prochain" (p) : un nœud qui a reçu l'identifiant de son voisin de gauche l'envoie vers son voisin de droite.
  - "gagnant" (g) : un nœud se déclare leader.

Un code d'amorçage est disponible a l'adresse `http ://www.lsv.fr/~hondet/resources/archos/ring.tar.gz`.

(a) Complétez la fonction «protocole» du fichier «ring-pipe.c» qui implémente le protocole sus-décrit.

Solution : ring-pipe crée plusieurs processus reliés par des pipes.

```
// délai moyen avant reception d'un message
#define DELAY 1.0

// les messages
#define MSG_VOISIN 'v'
#define MSG_PROCHAIN 'p'
#define MSG_GAGNANT 'g'

#include "ring-pipe.h"

// Comportement d'un nœud.
```

```
// Termine avec le numéro du gagnant comme code de sortie.

void protocole (int id)
{
        int actif = 1, nb;

        message("Mon identifiant est %d",id);

        envoyer(MSG_VOISIN,id);        // envoyer son identifiant au voisin

        while (1)
        {
                // accepter un message
                char c; int d;
                recevoir(&c,&d);

                if (!actif)
                {
                        // faire passer le message
                        envoyer(c,d);
                        if (c == MSG_GAGNANT)
                        {
                                message("nœud %d gagne",d);
                                exit(d);
                        }
                        continue;
                }

                if (c == MSG_VOISIN)
                {
                        nb = d;
                        message("id du voisin = %d",d);
                        envoyer(MSG_PROCHAIN,d);
                }
                else if (c == MSG_PROCHAIN)
                {
                        // évaluer les trois identifiants
                        message("ids = (%d,%d,%d)",id,nb,d);
                        if (d == id && nb >= d)
                        {
                                message("J'ai gagné !!");
                                envoyer(MSG_GAGNANT,id);
                        }
                        else if (nb > id && nb > d)
                        {
                                message("Je reste actif");
                                envoyer(MSG_VOISIN,id);
                        }
                        else
                        {
                                message("Je deviens passif");
```

```
                                actif = 0;
                        }
                }
                else if (c == MSG_GAGNANT)
                {
                        if (d != id)
                                message("erreur : quelqu'un a gagné alors "
                                        "que je suis actif?? (%d,%d)",d,id);
                        exit(id);
                }
        }
}

int main (int argc, char **argv)
{
        int i, n, status;

        if (argc != 2)
        {
                fprintf(stderr,"usage: ring <n>\n");
                exit(1);
        }

        // Générer les n clients. Chaque client est un processus fils
        // qui appelera protocole() avec les données dans struct node
        // préremplies.
        n = atoi(argv[1]);
        genere_noeuds(n);

        // On attend la terminaison de tout le monde pour affecter le gagnant.
        for (i = 0; i < n; i++) wait(&status);
        status = WEXITSTATUS(status);
        printf("[main] %d gagne\n",status);
        exit(status);
}
```

(b) Testez votre code avec plusieurs valeurs de n. Observez si votre programme termine toujours et s'il y a toujours un seul leader.

Solution : Au besoin réduire la valeur de DELAY pour n grand. Pour $n > 255$ le code de sortie (obtenu par `wait` ne suffit plus pour stocker l'identifiant du gagnant.

(c) La version réseau du programme, «ring-net.c» instancie un seul nœud par exécution. Vous pouvez recopier votre fonction «protocole» depuis «ring-pipe.c». Le programme prend 3 arguments : le port d'écoute du nœud, le nom du nœud voisin (nom de machine), le port de connexion au nœud voisin. Le programme crée un serveur dans un thread et attend la connexion d'un voisin. En parallèle, dans un autre thread, il tente une connexion sur son voisin (nom de machine et port passés en argument). Écrivez un script mettant en œuvre l'exécution de votre code sur 5 machines du département. Commencez par tester l'exécution de votre code localement (machine localhost en utilisant différents ports). Étendre progressivement la taille de l'anneau.

Solution :

```
// délai moyen avant reception d'un message
#define DELAY 1.0

// les messages
#define MSG_VOISIN 'v'
#define MSG_PROCHAIN 'p'
#define MSG_GAGNANT 'g'

#include "ring-net.h"

// Comportement d'un nœud.
// Termine avec le numéro du gagnant comme code de sortie.

void protocole (int id)
{
        int actif = 1, nb;

        message("Mon identifiant est %d",id);

        envoyer(MSG_VOISIN,id);        // envoyer son identifiant au voisin

        while (1)
        {
                // accepter un message
                char c; int d;
                recevoir(&c,&d);

                if (!actif)
                {
                        // faire passer le message
                        envoyer(c,d);
                        if (c == MSG_GAGNANT)
                        {
                                message("nœud %d gagne",d);
                                exit(d);
                        }
                        continue;
                }

                if (c == MSG_VOISIN)
                {
                        nb = d;
                        message("id du voisin = %d",d);
                        envoyer(MSG_PROCHAIN,d);
                }
                else if (c == MSG_PROCHAIN)
                {
                        // évaluer les trois identifiants
                        message("ids = (%d,%d,%d)",id,nb,d);
                        if (d == id && nb >= d)
```

```
                        {
                                message("J'ai gagné !!");
                                envoyer(MSG_GAGNANT,id);
                        }
                        else if (nb > id && nb > d)
                        {
                                message("Je reste actif");
                                envoyer(MSG_VOISIN,id);
                        }
                        else
                        {
                                message("Je deviens passif");
                                actif = 0;
                        }
                }
                else if (c == MSG_GAGNANT)
                {
                        if (d != id)
                                message("erreur : quelqu'un a gagné alors "
                                        "que je suis actif?? (%d,%d)",d,id);
                        exit(id);
                }
        }
}

// Les arguments qu'on donne sur la ligne de commande
int sock_server;
char *hostname;
int sock_client;

void* server_thread (void *arg)
{
        // accepter la connection de notre voisin à gauche
        node.fd_in = accept_connection(sock_server);
        return NULL;
}

void* client_thread (void *arg)
{
        // nous connecter à notre voisin à droite
        node.fd_out = connect_to_server(hostname,sock_client);
        return NULL;
}

int main (int argc, char **argv)
{
        pthread_t t1, t2;

        if (argc != 4)
        {
                fprintf(stderr,"usage: ring <sock1> <host> <sock2>\n");
```

```
                exit(1);
        }

        sock_server = atoi(argv[1]);
        hostname = argv[2];
        sock_client = atoi(argv[3]);

        pthread_create(&t1, NULL, server_thread, NULL);
        pthread_create(&t2, NULL, client_thread, NULL);
        pthread_join(t1,NULL);
        pthread_join(t2,NULL);

        // générer et lancer le nœud
        create_node();
        protocole(node.id);
        return 0;
}
```

(d) Créez un anneau avec vos voisins et l'étendre à toute la salle de TP.

Exercice 4 : Exclusion de Peterson.

Nous proposons d'implémenter l'algorithme d'exclusion mutuelle de Peterson décrit sur Wikipedia. Le principe consiste à utiliser 3 variables `f0`, `f1` et `turn` pour gérer l'entrée dans la «section critique» du code. La «section critique» est la section contenant le code à protéger des accès concurents. Prouvez que algorithme assure l'exclusion mutuelle.

(a) En vous basant sur la page Wikipedia de cet algorithme :

- modifiez la fonction `process0` et créez la fonction `process1` implémentant l'algorithme de Gary L. Peterson ; **Note :** `process0` et `process1` impléméntent la même fonctionnalité mais pour les threads 0 et 1.
- Testez votre code et commentez.

Solution : https ://www-ssl.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf Section 8-6 Vol. 3A 8.2.2 Memory Ordering in P6 and More Recent Processor Families

ou

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

#define MAX 100000000

int counter = 0;
char f0 = 0, f1 = 0, v = 0;

void* P0 (void* data)
{
        int i;
        for (i = 0; i < MAX; i++)
        {
```

```
                f0 = 1;                 // algorithme de Peterson pour P0
                v = 0;
                while (f1 && v == 0);

                // section critique
                counter++;

                f0 = 0;                 // on quitte la section critique
        }
        return NULL;
}

void* P1 (void* data)
{
        int i;
        for (i = 0; i < MAX; i++)
        {
                f1 = 1;                 // algorithme de Peterson pour P1
                v = 1;
                while (f0 && v == 1);

                // section critique
                counter++;

                f1 = 0;                 // on quitte la section critique
        }
        return NULL;
}

int main ()
{
        pthread_t t1;
        pthread_t t2;

        pthread_create(&t1, NULL, P0, NULL);        // create first thread
        pthread_create(&t2, NULL, P1,   NULL);        // create second thread

        pthread_join(t1, NULL);  // wait for first thread
        pthread_join(t2, NULL);  // wait for second thread

        printf("Counter: %d\n", counter);
        return 0;
}
```

(b) En vous basant sur le code de `process0` et `process1` écrire la fonction `process` qui permettra l'invocation générique d'un thread.

(c) Toujours en vous basant sur la page Wikipedia de l'algorithme de Peterson, modifiez votre code pour 4 processus.

(d) Mêmes questions que précédemment pour l'algorithme de Dekker.