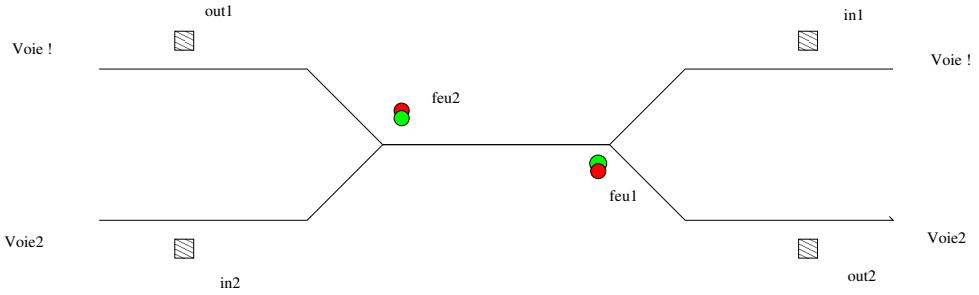
TD2/TME2 : Ordonnanceur et Threads (POSIX et `fair`)

Exercice 1 – (Modélisation) Chemin de Fer

Cet exercice est dédié à l'étude d'un problème de circulation de trains :



Généralement, une voie de chemin de fer est réservée pour permettre aux trains de rouler dans un seul sens et une autre voie parallèle à la première réservée pour permettre aux trains de rouler dans le sens opposé. Ces deux voies parallèles sont indiquées dans le dessin ci-dessus par `Voie1` et `Voie2`. Il arrive quelques fois, pour des raisons de place, que les deux voies soient regroupées en une seule obligeant ainsi des trains roulant en sens opposés de partager cette voie commune. Pour régler la circulation sur le tronçon à risque, les ingénieurs des chemins de fer disposent de deux types de dispositifs pour contrôler le croisement des trains :

- des feux qui peuvent être soit `VERT`, soit `ROUGE`,
- des détecteurs de présence qui annoncent la présence (`ALLUME`) ou non (`ETEINT`) d'un train sur le tronçon.

Question 1

Enumérer l'ensemble des dispositifs nécessaires pour le croisement décrit ci-dessus. On associera à chaque dispositif un nom de variable et les valeurs que peut prendre cette variable.

Question 2

Quel est le nombre d'états possibles du système global ?

Une propriété de sûreté (*safety* $\neq$ *liveness*) d'un système doit être vérifiée durant toute exécution du système. Un problème de sûreté est donc une condition qui doit être fausse durant toute exécution du système. De manière générale, on découpe les exécutions en *états* et on teste que les états vérifiants la propriété de sûreté sont inatteignables.

Deux types de problème de sûreté peuvent apparaître :

- les problèmes de cohérence qui correspondent à des états incohérents (ici dangereux) du système,
- et les problèmes d'inter-blocage.

Question 3

En utilisant les variables introduites précédement, exprimer (au moins) deux problèmes de cohérence et un problème d'inter-blocage pour le croisement.

Une propriété de *vivacité* du système doit inévitablement être vérifiée à un certain moment de l'exécution du système.

Question 4

Identifier un problème de vivacité dans le croisement de train.

Exercice 2 – Rappels sur les Fair Threads

Question 1
Rappeler la différence entre *fair threads* et *threads posix*.

Question 2

1. (Cours) Ecrire un programme contenant quatre fonctions similaires répétant les deux actions suivantes :
   – écrire sur la sortie erreur la même partie de la phrase [1] de monsieur Jourdain *"Belle marquise / vos beaux yeux / me font mourir / d'amour"*.
   – rendre la main au scheduler.
   Puis, en utilisant les Fair Threads, lancer quatre threads exécutant les quatre fonctions dans le même scheduler.
2. Ecrire le même programme, mais en lancant les quatre threads dans quatre schedulers différents.
3. Que va t-il se passer à l'exécution ? Pourquoi ?

Exercice 3 – (POSIX) Attentes actives

Question 1
Ecrire un programme architecturé autour de deux threads :
– un thread *requête* qui demande en permanence un entier `n` à l'utilisateur et
– un thread *lecteur* qui va lire dans `/dev/urandom` `n` entiers et les affiche.

Le thread *lecteur* communiquera *ici* avec le thread de requête par l'intermédiaire d'une variable globale (partagée) `n` dont le changement de valeur déclenchera la lecture. Utiliser d'abord des mutex pour protéger les manipulations sur `n`.

Question 2

Que remarque t-on à l'exécution du programme, sur le plan de l'efficacité ? Utiliser les variables de condition des threads POSIX pour éviter ce gâchis de ressources.

Exercice 4 – (Fair Threads) Envoi/Attente

On propose dans cet exercice de traiter avec les *fair threads* un mécanisme d'envoi et d'attente d'événements qui peuvent se produire au même instant. (voir `td2_2.canevas` dans Annexe).

Question 1
Compléter la boucle de la procédure **`void`** `awaiter(`**`void`** `*args)` donnée pour :
– attendre un événement,
– imprimer "Evénement recu" à la réception de l'événement,
– redonner le contrôle à l'ordonnanceur pour 3 instants,
– une fois les 5 événements reçus, arrêter le thread générateur d'événements.

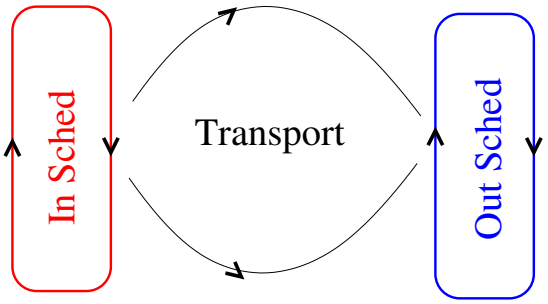
Question 2
Compléter la boucle de la procédure **`void`** `generator (`**`void`** `*args)` donnée pour
– redonner le contrôle à l'ordonnanceur pour 7 instants,
– imprimer "Evénement numéro compteur_de_boucle envoyé",
– générer l'événement `evt`.

1. classique dans les exercices sur les threads

Question 3
Dans le `main()`, créer l'ordonnanceur `sched`, l'événement `evt`, le thread `ft_trace` (le traceur d'instant), le thread `ft_generator` (le générateur d'événements) et le thread `ft_awaiter` (qui attend l'événement `evt`).

Exercice 5 – (Fair Threads) Producteur/Consommateur logistique et fair threads

Dans cet exercice nous revisitons le patron Producteur/Consommateur et nous nous placerons dans le cas où le producteur et le consommateur s'ignorent et se comportent d'une manière indépendante (chacun son propre ordonnanceur `in_sched` et `out_sched`). Dans une boucle, le producteur créée un produit et le range dans sa file `in`. Sous peine de collision avec le producteur, le consommateur, pour prendre un produit, n'accède pas à la file `in` du producteur, mais plutôt à sa propre file `out`. Continuellement, il y a des intermédiaires `thread_array[MAX_THREADS]` qui se chargent de transporter les produits de la file `in` vers la file `out` sans provoquer de collisions avec le producteur et le consommateur.



Avec l'aide du canevas `prodconsft.canevas` (voir Annexe),

Question 1
Compléter la boucle de la procédure `void produce(void *args)` pour permettre au producteur de ranger un produit dans la file `in`.

Question 2
Compléter la boucle de la procédure `void consume(void *args)` pour permettre au consommateur de retirer un produit de la file `out`.

Question 3
Compléter la boucle de la procédure `void process_value (int v, int n)` pour que les intermédiaires `thread_array[MAX_THREADS]` puissent transporter les produits de la file `in` vers la file `out` sans collision avec le producteur, ni avec le consommateur. On rappelle que le producteur et le consommateur sont sur deux ordonnanceurs différents.

Exercice 6 – (Fair Threads) Automate

A l'aide du canevas `automate.canevas`,
Ecrire un automate à 3 états :
- état 0 : imprime à la sortie "Begin",
- état 1 : attend l'événement `evnt1` et imprime à la sortie "Hello" à sa réception,
- état 2 : attend l'événement `evnt2` et imprime à la sortie "World" à sa réception et retourne à l'état 1.

Question 1
Compléter la procédure `generator()` pour envoyer successivement l'événement `evnt1`, puis l'événement `evnt2` en coopérant avec l'automate.

Annexe

```
/****************************** td2_2.canevas ******************************/

#include "fthread.h"
#include "stdio.h"
#include "unistd.h"
#include "traceinstantsf.h"

ft_event_t  evt;

void awaiter (void *args)
{
  int i, res;

  for (i = 0; i< 5; i++) {
    fprintf(stdout, "debut awater\n");
    /* ?????????? */
  }
  fprintf(stdout, "stop generator!\n");
  /* ?????????? */
}

void generator (void *args)
{
  int i;

  for (i=0;; ++i) {
    /* ?????????? */
  }
}



int main (void)
{
  ft_thread_t ft_trace, ft_awaiter, ft_generator;
  ft_scheduler_t sched = ft_scheduler_create ();

  /* ?????????? */

  ft_scheduler_start(sched);


  ft_exit ();
  return 0;
}

#define PRODUCED 30             // number of produced values
#define FILE_SIZE 20            // size of files
#define MAX_THREADS 5           // number of threads
#define PROCESSING 150000000    // processing delay

#define PRINT(s,v) fprintf (stderr,s,v)

/****************************** prodconsft.canevas ************************/

#include "fthread.h"
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>
#include "prodconsft.h"
#include "traceinstantsf.h"

/************************************/
typedef struct cell
{
   int          value;
   struct cell *next;
} *cell;

typedef struct file
{
   int  length;
   cell first;
   cell last;
} *file;

file add (int v, file l)
{
   cell c   = (cell)malloc(sizeof(struct cell));
   c->value = v;
   c->next  = NULL;

   if (l == NULL) {
      l         = (file)malloc(sizeof(struct file));
      l->length = 0;
      l->first  = c;
   } else {
      l->last->next = c;
   }
   l->length++;
   l->last = c;
   return l;
}

void put (int v, file *l)
{
  (*l) = add(v,*l);
}

int get (file *l)
{
  int res;
  file f = *l;
  if (l == NULL) {
     fprintf (stdout, "get error!\n");
     return 0;
  }
  res = f->first->value;
  f->length--;
  if (f->last == f->first) {
     *l = NULL;                  //     a la place de f = NULL;
  } else {
    f->first = f->first->next;
  }
  return res;
}
```

```
int size (file l)
{
   if (l == NULL) return 0;
   return l->length;
}


/************************************/

file in = NULL, out = NULL;
ft_scheduler_t in_sched, out_sched;
ft_event_t new_input, new_output;

/************************************/


void process_value (int v, int n)
{
  int i, j;

  /* ?????????? */

}

void process (void *args)
{
  while (1) {
    if (size(in) > 0) {
       process_value(get(&in), (int)args);
    } else {
       ft_thread_await (new_input);
       if (size(in) == 0) ft_thread_cooperate();
    }
  }
}

void produce (void *args)
{
  int v = 0;
  while (v < PRODUCED) {

    /* ?????????? */

  }
}

void consume (void *args)
{
  int v = 0;
  while (v < PRODUCED) {

    /* ?????????? */

  }
  exit(0);
}

/************************************/
```

```
int main (void)
{
   int i;
   ft_thread_t thread_array[MAX_THREADS];

   in_sched   = ft_scheduler_create();
   out_sched  = ft_scheduler_create();

   new_input  = ft_event_create(in_sched);
   new_output = ft_event_create(out_sched);

   ft_thread_create(in_sched, traceinstants, NULL, (void *)50);

   for (i=0; i<MAX_THREADS; i++) {
       thread_array[i] = ft_thread_create(in_sched, process, NULL, (void *)i);
   }

   ft_thread_create(in_sched, produce, NULL, NULL);
   ft_thread_create(out_sched, consume, NULL, NULL);

   ft_scheduler_start(in_sched);
   ft_scheduler_start(out_sched);

   ft_exit();
   return 0;
}

/****************************** automate.canevas ****************************/

#include "fthread.h"
#include "stdio.h"
#include "unistd.h"
#include "traceinstantsf.h"

ft_event_t  event1, event2;

/* ?????????? */

void generator (void *args)
{
  int i;

  for (i=0; i < 15; ++i) {
      /* ?????????? */
  }
}


int main ()
{
  ft_scheduler_t sched = ft_scheduler_create();
  event1 = ft_event_create(sched);
  event2 = ft_event_create(sched);

  ft_thread_create(sched, traceinstants, NULL, (void *)15);
  if (NULL == ft_automaton_create(sched, autom, NULL, NULL)) {
     fprintf(stdout, "Cannot create automaton!!!\n");
  }
```

```
  ft_thread_create(sched, generator, NULL, NULL);

  ft_scheduler_start(sched);


  ft_exit();
  return 0;
}
```