Informations sur le cours PC2R

Site

https://www-master.ufr-info-p6.jussieu.fr/2009/

http://www-apr.lip6.fr/~chaillou/Public/enseignement/2009-2010/pc2r/

Equipe pédagogique

- cours : **Emmanuel Chailloux**
- TD/TME groupes 1
  Philippe Trébuchet
- TD/TME groupe 1bis (mercredi après-midi) :
  Philippe Wang, Tong Lieu, Benjamin Canou
- TD/TME groupe 2 (vendredi après-midi) :
  Tong Lieu

Logiciels du cours

Installation ARI :

- O'Caml 3-10 (préinstallé)
- Java 1.6 (préinstallé)
- GCC 4.1.2 (préinstallé)

Installation dans : `/users/Enseignants/chaillou/usr/local/`

- FTthread pour C
- Esterel

Sources dans `/users/Enseignants/chaillou/install`

Rappel du cours 1

- parallélisme : perte du déterminisme
- Modèles de parallélisme
  - mémoire partagée :
    synchronisation explicite/communication implicite
  - mémoire répartie :
    synchronisation implicite/communication explicite

  $\Rightarrow$ dualité des deux modèles

Cours 2 : Threads équitables

1. Généralités : coopération vs préemption
2. Api Fairthreads en C : scheduler et threads
3. Implantation
4. Evénements
5. Automates

Fair Threads

- Frédéric Boussinot
- projet MIMOSA EMP-CMA / Inria Sophia Antipolis sur la programmation réactive :

  `http://www-sop.inria.fr/mimosa/rp`
- Fair Threads :

  `http://www-sop.inria.fr/mimosa/rp/FairThreads/`

Modèle coopératif et préemptif

- ordonnanceur (*scheduler*) : serveur de synchronisation
- 2 types de threads
  - threads liés à un ordonnanceur (modèle coopératif)
  - threads non liés (modèle préemptif)

Caractéristiques

- multiprocesseurs : schedulers et threads non liés;
- déterministe : si tous les threads sont liés à un seul scheduler;
- I/O bloquantes : implantées par threads non liés;

Caractéristiques (suite)

- *instant* : partagé par tous les threads d'un scheduler; synchronisation automatique à la fin de chaque instant
- *événement* : diffusion instantanée à tous les threads liés à un même scheduler;
  permet la synchronisation et la communication
- *automate* : pour les petits threads de courte vie; implantation légère

Schedulers

- serveur de synchronisation (instants)
- serveur de communication (événements)
- serveur d’exécution (automates)

Ordonnancement coopératif

Durant un instant :

- exécution de chaque thread jusqu'au prochain point de coopération :

Un thread rend la main au scheduler à un point de coopération :

- explicite : fonction `cooperate`
- implicite : attente d'un événement
- pas de priorité entre threads d'un même scheduler

Ordonnancement préemptif

- modèle à mémoire partagée
- perte du déterminisme
- mutuelle exclusion (Mutex)
- attente sur condition (Condition)

prochain cours : Thread en O’Caml

Automates

petit thread ne nécessitant pas une pile propre
contient une liste d'états (code séquentiel)

- s'exécute dans le thread du scheduler
- effectue un changement d'état en un instant
- passage d'un état à un autre :
  - explicite : saut à un état particulier
  - implicite : passage à l'état suivant
- fin de l'automate, à la fin du dernier état
- peut communiquer par événement (état particulier)

Evénements

- création et diffusion d'un événement à tous les trheads
- attente d'un événement à un instant ou au plus sur n instants
- association d'une valeur à un événement pour un instant et récupération de celle-ci
- sélection sur un tableau d'événements

Implantation des Fair Threads

- en C :

  `http://www-sop.inria.fr/mimosa/rp/FairThreads/FTC/index.html`

- en Java :

  `http://www-sop.inria.fr/mimosa/rp/FairThreads/FTJava/index.html`

- en Scheme :

  `http://www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo-8.html`

- en O'Caml (prototype 1) :

  `http://www-apr.lip6.fr/~chaillou/Public/Dev/HirondML/`

Bibliothèque C : scheduler

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

- type `ft_scheduler_t`
- création : `ft_scheduler_t ft_scheduler_create (void)`

  retourne `NULL` si échec de la création
- démarrage : `int ft_scheduler_start (ft_scheduler_t sched)`

  code retour 0 ou un code d'erreur $\neq 0$ (`BAD_CREATE`)

Bibliothèque C : scheduler (suite)

Contrôle des threads :

- `int ft_scheduler_stop    (ft_thread_t th)`
  force l'arrêt du thread `th`
- `int ft_scheduler_suspend (ft_thread_t th)`
  suspend l'exécution du thread `th` au prochain instant
- `int ft_scheduler_resume  (ft_thread_t th)`
  reprend l'exécution du thread `th` au prochain instant

La suspension est prioritaire à la reprise.

Bibliothèque C : thread

- type `ft_thread_t`
- `ft_thread_t ft_thread_create (ft_scheduler_t sched, void (*runnable)(void*), void (*cleanup)(void*), void *args)`
  où
  - `sched` : scheduler
  - `runnable` : fonction de calcul du thread
  - `cleanup` : fonction de nettoyage
  - `args` : argument des 2 fonctions

Bibliothèque C : thread (suite)

Fin d'un thread :

- fin du calcul de la fonction associée
- appel à `void ft_exit (void)`
- appel à `int ft_scheduler_stop (ft_thread_t th)`

Quand un thread termine la fonction `cleanup` est appelée à l'instant suivant

Bibliothèque C : thread (fin)

Attente de fin d'un thread :

- `int ft_thread_join (ft_thread_t th)`

  attente de la fin du thread `th`

- `int ft_thread_join_n (ft_thread_t th,int n)`

  attente sur au plus n instants

Coopération

- `int ft_thread_cooperate (void)`
  retourne le contrôle au scheduler
- `int ft_thread_cooperate_n (int n)`
  redonne le contrôle pour n instants

Equivalent à :

```
for (i=0;i<k;i++) ft_thread_cooperate ()
```

Un premier exemple : Hello World (1)

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

void h (void *id)
{
   while (1) {
      fprintf (stderr,"Hello ");
      ft_thread_cooperate ();
   }
}
void w (void *id)
{
   while (1) {
      fprintf (stderr,"World!\n");
      ft_thread_cooperate ();
   }
}
```

Un premier exemple : Hello World (2)

```
int main(void)
{
  ft_scheduler_t sched = ft_scheduler_create ();
  ft_thread_create (sched,h,NULL,NULL);
  ft_thread_create (sched,w,NULL,NULL);
  ft_scheduler_start (sched);

  ft_exit ();
  return 0;
}
```

Le même en non-déterministe

```
int main (void)
{ ft_scheduler_t sched1 = ft_scheduler_create ();
  ft_scheduler_t sched2 = ft_scheduler_create ();

  ft_thread_create (sched1,h,NULL,NULL);
  ft_thread_create (sched2,w,NULL,NULL);

  ft_scheduler_start (sched1);
  ft_scheduler_start (sched2);

  ft_exit ();
  return 0;
}
```

Liaison des threads

- `int ft_thread_unlink (void);`
  délie le thread de son scheduler
- `int ft_thread_link   (ft_scheduler_t sched);`
  relie un thread auprès du scheduler `sched`

permet de changer de scheduler.

Lecture non bloquante (1)

```
gcc -Wall -O3 -D_REENTRANT -I ../include -L../lib \
  nbread.c -lfthread -lpthread
#include "fthread.h"
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/*****************************************/
ssize_t ft_thread_read (int fd,void *buf,size_t count)
{
   ft_scheduler_t sched = ft_thread_scheduler ();
   ssize_t res;

   ft_thread_unlink ();
   res = read (fd,buf,count);
   ft_thread_link (sched);
   return res;
}
```

Lecture non bloquante (2)

```
/*****************************************/
void reading_behav (void* args)
{
   int max = (int)args;
   char *buf = (char*)malloc (max+1);
   ssize_t res;
   fprintf (stderr,"enter %d characters:\n",max);

   res = ft_thread_read (0,buf,max);

   if (-1 == res) fprintf (stderr,"error\n");
   buf[res] = 0;
   fprintf (stderr,"read %d: <%s>\n",res,buf);
   exit (0);
}
```

Lecture non bloquante (3)

```
int main (void)
{
  ft_scheduler_t sched = ft_scheduler_create ();
  ft_thread_create (sched,reading_behav,NULL,(void*)5);
  ft_scheduler_start (sched);
  ft_exit();
  return 0;
}
```

Implantation

- Utilise les threads POSIX (man pthread)
- environ 1800 lignes de C

Implantation (suite)

```
struct ft_scheduler_t
{ ft_thread_t              self;
  thread_list_t            thread_table;
  thread_list_t            to_run;
  thread_list_t            to_stop;
  thread_list_t            to_suspend;
  thread_list_t            to_resume;
  thread_list_t            to_unlink;
  broadcast_list_t         to_broadcast;
  pthread_mutex_t          sleeping;
  pthread_cond_t           awake;
  ft_environment_t         environment;
  int                      well_created; };
```

Implantation (suite)

```
struct ft_thread_t {
   pthread_t                pthread;
   int                      well_created;
   pthread_mutex_t          lock;
   pthread_cond_t           token;
   int                      has_token;

   ft_executable_t          cleanup;
   ft_executable_t          run;
   void                    *args;

   ft_scheduler_t           scheduler;
...};
```

Implantation (suite)

```
static void _fire_all_threads (ft_scheduler_t sched)
{
   FOR_ALL_THREADS
   if (_is_fireable (thread)){
      if (!_is_automaton (thread)) {
        _transmit_token (sched->self,thread);
      } else {
        _run_as_automaton (thread);
      }
   }
   END_FOR_ALL
}
```

Evénements

- type `ft_event_t`
- création : `ft_event_t ft_event_create (ft_scheduler_t sched);`

A l'instant courant :

- génération : `int ft_thread_generate (ft_event_t evt);`
  engendre l'événement `evt` pour l'instant courant; il aura disparu à l'instant suivant
- `int ft_thread_generate_value (ft_event_t evt,void *val);`
  associe une valeur `val` à la génération de l'événement

Evénements (suite)

A l'instant suivant

- `int ft_scheduler_broadcast (ft_event_t evt);`
  l'événement `evt` sera engendré au prochain instant
- `int ft_scheduler_broadcast_value (ft_event_t evt,void *val);`
  `val` est associée à `evt`

Attente d’un événement

- attente
  - `int ft_thread_await (ft_event_t evt);`
    suspend l’exécution du thread jusqu’à la génération d’`evt`
  - `int ft_thread_await_n (ft_event_t evt,int n);`
    l’attente dure au plus n instants.
- récupération d’une valeur :
  - `ft_thread_get_value(ft_event e, int num, void **result)`
    récupère la i-ième valeur associée à l’événement e :
    - si elle existe, la valeur est rangée dans result, l’appel termine immédiatement
    - sinon, la fonction retourne `NULL` à l’instant suivant

Attente sur tableau d'événements

permet l'attente sur plusieurs événements.
Le tableau d'événements `array` et le tableau `mask` sont de de longueur len.

- `int ft_thread_select(int len,ft_event_t *array,int *mask)`
  suspend l'exécution du thread jusqu'à la génération d'au moins un événement du tableau `array`; le tableau `mask` indique quels sont les événements engendrés.
- ```
  int ft_thread_select_n (int len,ft_event_t *array,
                          int *mask,int timeout);
  ```
  Attente au plus $timeout$ instants
  ```

Exemple avec événements (1)

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

ft_event_t e1, e2;

void behav1 (void *args)
{
  ft_thread_generate (e1);
  fprintf (stdout,"broadcast e1\n");

  fprintf (stdout,"wait e2\n");
  ft_thread_await (e2);
  fprintf (stdout,"receive e2\n");

  fprintf (stdout,"end of behav1\n");
}
```

Exemple avec événements (2)

```
void behav2 (void *args)
{
  fprintf (stdout,"wait e1\n");
  ft_thread_await (e1);
  fprintf (stdout,"receive e1\n");

  ft_thread_generate (e2);
  fprintf (stdout,"broadcast e2\n");

  fprintf (stdout,"end of behav2\n");
}
```

Exemple avec événements (3)

```
int main(void)
{
   int c, *cell = &c;
   ft_thread_t th1, th2;
   ft_scheduler_t sched = ft_scheduler_create ();

   e1 = ft_event_create (sched);
   e2 = ft_event_create (sched);

   th1 = ft_thread_create (sched,behav1,NULL,NULL);
   th2 = ft_thread_create (sched,behav2,NULL,NULL);

   ft_scheduler_start (sched);

   pthread_join (ft_pthread (th1),(void**)&cell);
   pthread_join (ft_pthread (th2),(void**)&cell);
   fprintf (stdout,"exit\n");
   exit (0);
}
```

Exemple avec événements (4)

```
/*
broadcast e1
wait e2
wait e1
receive e1
broadcast e2
end of behav2
receive e2
end of behav1
exit
*/
```

Tableau d'événements (1)

```
/* use of select to await 2 events */

ft_event_t  a,b;

void awaiter (void *args)
{
   ft_event_t events [2] = {a,b};
   int        result [2] = {0,0};

   ft_thread_select (2,events,result);
   fprintf (stdout, "result: [%d,%d] ",result[0],result[1]);
   if (result[0] == 0 || result[1] == 0) {
      ft_thread_await (result[0]==0 ? events[0] : events[1]);
   }

   fprintf (stdout, "both received! ");
   ft_thread_cooperate ();
   fprintf (stdout, "exit!\n");
   exit (0);
}
```

Tableau d’événements (2)

```
void trace_instant (void *args)
{
   int i = 1;
   while (1) {
      fprintf (stdout, "\ninstant %d: ",i);
      i++;
      ft_thread_cooperate ();
   }
}
```

Tableau d’événements (3)

```
void agenerator (void *args)
{
   ft_thread_cooperate_n (3);
   fprintf (stdout, "event a generated! ");
   ft_thread_generate (a);
}

void bgenerator (void *args)
{
   ft_thread_cooperate_n (3);
   fprintf (stdout, "event b generated! ");
   ft_thread_generate (b);
}
```

Tableau d'événements (4)

```
int main (void)
{
   ft_scheduler_t sched = ft_scheduler_create ();

   a = ft_event_create (sched);
   b = ft_event_create (sched);
   ft_thread_create (sched,trace_instant,NULL,NULL);

   ft_thread_create (sched,agenerator,NULL,NULL);
   ft_thread_create (sched,awaiter,NULL,NULL);
   ft_thread_create (sched,bgenerator,NULL,NULL);

   ft_scheduler_start (sched);

   ft_exit ();
   return 0;
}
```

Tableau d’événements (5)

```
/* result

instant 1:
instant 2:
instant 3:
instant 4: event a generated! result: [1,0] event b generated! both received
instant 5: exit!
end result */
```

Automates

Ensemble de macros permettant de décrire les états d'un automate et les attentes sur événements.

Création :

```
ft_thread_t ft_automaton_create (ft_scheduler_t sched,
                                 ft_automaton_t automaton,
                                 ft_executable_t cleanup,
                                 void *args)
```

attente sur événement avec automate (1)

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

/* simultaneous events */

ft_event_t event1,event2;

DEFINE_AUTOMATON (autom)
{
   BEGIN_AUTOMATON
      STATE_AWAIT (0,event1);
      STATE_AWAIT (1,event2)
      {
         fprintf (stdout, "both events are received! ");
      }
   END_AUTOMATON
}
```

attente sur événement avec automate (2)

```
void generator (void *args)
{
   ft_thread_cooperate_n (4);
   fprintf (stdout, "event1 generated! ");
   ft_thread_generate (event1);

   ft_thread_cooperate_n (4);
   fprintf (stdout, "event1 and event2 are generated! ");
   ft_thread_generate (event1);
   ft_thread_generate (event2);

   ft_thread_cooperate ();
   fprintf (stdout, "exit\n");
   exit (0);
}
```

attente sur événement avec automate (3)

```
void traceInstants (void *args)
{
   int i = 0;
   for (i=0;i<10;i++) {
      fprintf(stdout,"\n>>>>>>>>>>> instant %d: ",i);
      ft_thread_cooperate ();
   }
   fprintf (stdout, "exit!\n");
   exit (0);
}
```

attente sur événement avec automate (4)

```
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,NULL);

   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;
}
```

attente sur événement avec automate (5)

```
/* result

>>>>>>>>>>> instant 0:
>>>>>>>>>>> instant 1:
>>>>>>>>>>> instant 2:
>>>>>>>>>>> instant 3:
>>>>>>>>>>> instant 4: event1 generated!
>>>>>>>>>>> instant 5:
>>>>>>>>>>> instant 6:
>>>>>>>>>>> instant 7:
>>>>>>>>>>> instant 8: event1 and event2 are generated! both events are rece
>>>>>>>>>>> instant 9: exit
end result */
```

API FT pour O'Caml

- TER puis projet migration de threads
  - HirondML :

    `http://www-apr.lip6.fr/~chaillou/Public/Dev/HirondML/`
  - sémantique différente sur :
    - l'envoi/réception d'événements
  - pas de préemptif (sauf thread Caml)
  - mais bibliothèque spéciale pour les I/O
- Master STL :
  - HirondML 2: respect de la sémantique des FT

Une implémentation pour OCaml

- Première implémentation : Une surcouche des threads OCaml
  - Un scheduler est un jeton
  - Synchronisation à base de conditions
  - Détachement de fair thread « facile »
- Les problèmes
  - Implémentation simple mais peu efficace
  - Surcouche des threads OCaml pour l'implémentation des threads détachés

les threads détachés sont surtout utilisés pour des IO bloquantes

Idée : une séparation des tâches

- Une librairie pour effectuer des actions bloquantes
- Simuler la concurrence des schedulers
- Nouvelle donne
  - La réactivité d'un scheduler n'est plus assurée, mais on peut mesurer les schedulers « trop lent » facilement par une trace
  - On perd la couche POSIX

C'est l'implémentation actuelle.

FT : Exemple1

```
let sched=Fthread.create_scheduler();;

let rec fth x=
  Printf.printf "je suis le ft %d\n" x;
  Fthread.cooperate();
  fth x
;;

Fthread.create sched fth 1;
Fthread.create sched fth 2;
Fthread.start_scheduler sched;
Fthread.exit();;

(* affichage de fth 1 | fth2 ----> *)
(* je suis le ft 1
   je suis le ft 2
   je suis le ft 1
   ... *)
```

FT : Compilation exemple 1

sur ari-31-312-01

```
$ ocamlopt -c -I ../lib exfthread.ml
$ ocamlopt -I ../lib unix.cmxa fthread.cmxa \
   exfthread.cmx -o exfthread.exe
$ ./exfthread.exe
je suis le ft 1
je suis le ft 2
je suis le ft 1
je suis le ft 2
je suis le ft 1
je suis le ft 2
...
```

où ../lib correspond à :

```
/users/Enseignants/chaillou/install/migrate-0.3/lib
```

FT : Exemple 2 (1)

```
let table1=Fthread.create_scheduler();;
let table2=Fthread.create_scheduler();;
let finale=Fthread.create_scheduler();;

let nbr_vainqueur=ref 0;;
let finale_commencee=ref false;;
let passe=Fthread.create_event();;
...
for i=1 to 4 do
  Fthread.create table1 joueur (i,[1;0;2;4;0;3]);
  Fthread.create table2 joueur ((i+5),[2;1;0;4;3])
done;

Fthread.create table1 joueur (5,[1;2;8;4;4;3;2;8]);
Fthread.create table2 joueur (10,[1;8;3;3;2;8]);
Fthread.start_scheduler table1;
Fthread.start_scheduler table2;
Fthread.exit();;
```

FT : Exemple 2 (2)

```
let rec joueur (id,cartes)=
  match cartes with
      c :: rc ->
        if c=0 then (... Fthread.awaitn 1 passe;  joueur (id,rc))
        else if c>7 then (...    incr nbr_vainqueur; Fthread.link_to finale;
                              joueur (id, rc))
        else (... Fthread.cooperate(); joueur (id,rc))
   | _ -> (... if !nbr_vainqueur = 2 then (
             nbr_vainqueur:=0; Fthread.start_scheduler finale)
```

Bibliographie

- Boussinot, F. – Java Fair Threads – Inria research report, RR-4139, 2001.
- Boussinot, F. – FairThreads: mixing cooperative and preemptive threads in C – Inria research report, RR-5039, December, 2003.
- Serrano, M. et Boussinot, F. et Serpette, B. – Scheme Fair Threads – 6th sigplan International Conference on Principles and Practice of Declarative Programming (PPDP), Verona, Italy, Aug, 2004, pp. 203–214.
- Chailloux, E. et Ravet, V. et Verlaguet, J. — HirondML: Fair Threads Migrations for Objective Caml — Parallel Processing Letters, volume=18-1, 2008.