

PC3R

## Cours 02 - Modèles Concurrents - Coopération

Romain Demangeon

PC3R MU1IN507 - STL S2

30/01/2023

# Plan du Cours 2

- ▶ Modèles de concurrence.
- ▶ Modèle coopératif: *Fair Threads*
- ▶ Modèle coopératif: *Lwt*

# Modèles de Concurrence: Système

- ▶ le **système d'exploitation** gère la concurrence à l'aide d'un **ordonnanceur** responsable de l'allocation des tâches (processus / threads) sur les unités de calcul physique.
- ▶ les **processus** ne partagent pas de mémoire virtuelle:
  - ▶ communication à travers des **fichiers**,
  - ▶ communication dans espace **mémoire** partagé,
  - ▶ communication par **mécanismes** spécifiques: **signaux** (signaux POSIX), **sockets** (*Unix Domain Socket*), **pipelines**.
- ▶ les **threads** système (processus légers) partagent leur **tas** (mais, bien sûr, pas la pile):
  - ▶ écueils habituels de la **concurrence**,
  - ▶ **communication** directe (par la mémoire),
  - ▶ accès à des **mécanismes** de communication et synchronisation,
- ▶ **intérêt** des **threads**:
  - ▶ le **changement de contexte** (lors de la préemption/élection) est beaucoup moins couteux,
  - ▶ le **partage** n'exige pas d'API spécifique
- ▶ **coût** de la programmation multi-thread:
  - ▶ **difficulté** de la programmation: écueils de la concurrence.
  - ▶ **limites**: les **piles** s'accumulent en mémoire.

# Modèles de Concurrence: Préemption ou Coopération

## Dualité préemption/coopération:

- ▶ **préemption**: l'ordonnanceur stoppe lui-même une tâche en cours d'exécution et en élit une autre,
- ▶ **coopération**: l'ordonnanceur attend un signal explicite de la tâche pour la stopper et en élire une autre.
- ▶ **avantages comparés**:
  - ▶ la préemption est plus **naturelle** (pas besoin d'expliciter les opérations les opérations de scheduling)
  - ▶ la préemption est plus **difficile** à programmer (écueils).
- ▶ **en pratique**: la **préemption** est utilisée dans les systèmes d'exploitation et la plupart des langages.

# Modèles de Concurrence: Modèle de Thread

- ▶ Threads *systèmes/noyaux* (*kernel-level threads*):
  - ▶ threads gérés *directement* par le système d'exploitation
  - ▶ partagent l'*ordonnanceur* du système (avec les processus).
- ▶ Threads *application/utilisateur* (*user-level threads*):
  - ▶ threads gérés par l'*application* (le *runtime* du langage)
  - ▶ l'*ordonnanceur* est *propre* à l'*application*
    - ▶ il peut être *préemptif* ou *coopératif*
    - ▶ il ne gère pas d'*autres* threads/processus.
- ▶ Avantages comparés:
  - ▶ *systèmes*: utilisation implicite de l'*ordonnancement multi-coeur*.
  - ▶ *utilisateurs*: facilité et *rapidité* de manipulation
  - ▶ *utilisateurs*: *portabilité*

# Modèles de Concurrence: Modèle de Thread (II)

- ▶ Différents langages (en fait, différents environnements d'exécution, possiblement d'un même langage) utilisent différents modèles de multithreading.
- ▶ Modèle 1:1
  - ▶ la création d'un thread par une opération du langage engendre la création d'un thread système.
  - ▶ le thread est géré par l'ordonnanceur du système.
- ▶ Modèle n:1
  - ▶ les threads sont gérés par l'environnement d'exécution
  - ▶ l'ordonnanceur dépend de l'environnement d'exécution (coopération possible)
- ▶ Modèle n:m
  - ▶ compromis entre les deux: les threads utilisateurs sont répartis entre plusieurs threads système.

# Modèles de Concurrence: Modèle de Thread (III)

- ▶ Opérations usuelles des API de programmation avec threads:
  - ▶ déclaration: les threads ont un type spécifique manipulés par les primitives
    - ▶ permet de manipuler explicitement un thread depuis un autre (l'attendre, le détruire, ...)
  - ▶ création: opération qui produit un nouveau thread
    - ▶ on peut passer au thread en paramètre, la tâche qu'il doit exécuter,
    - ▶ de manière duale, dans les langages objets, la tâche elle-même hérite de la classe des threads.
    - ▶ parfois séparée du démarrage (*run, start*) du thread lui-même.
  - ▶ attente: opération qui permet d'attendre la terminaison d'un thread, et donc effectuer une synchronisation.
    - ▶ souvent, aussi une attente temporisée.

# Modèles de Concurrence: Modèle Mémoire

- ▶ Opérations usuelles des API de programmation avec threads impliquant la mémoire:
  - ▶ mutex: verrous accordant un droit dont l'implémentation garantit qu'il ne peut être donné qu'à un seul thread.
    - ▶ mécanisme permettant d'éviter les compétitions.
  - ▶ conditions: opérations, liées aux verrous, qui permettent d'endormir (il n'est plus élu) et de réveiller (il peut être élu à nouveau) explicitement des threads.
    - ▶ mécanisme permettant d'éviter les attentes actives.
  - ▶ autres mécanismes: synchronisations explicites, primitives de coopération, communication explicites (signaux, évènements)

# Threads POSIX

- ▶ API **courante** de manipulation de threads 1:1,
  - ▶ bibliothèque en **C**,
  - ▶ **utilisée** pour l'implémentation des threads d'autres langages/bibliothèques.
  - ▶ implémentée par la plupart des systèmes Unix.
- ▶ **type** `p_thread_t`
- ▶ **créés** `pthread_create (thread,attr,start_routine,arg)`
  - ▶ un **identifiant thread**,
  - ▶ des **attributs**,
  - ▶ une **routine** (fonction) et ses **arguments**.
- ▶ un système de **mutexes**
  - ▶ `pthread_mutex_lock(mut)` bloque si `mut` est pris par un autre thread.
- ▶ un système de **conditions**
  - ▶ `pthread_cond_wait(condi, mut)` relâche le mutex `mut`, endort le thread jusqu'à ce que `condi` soit signalée.

# Threads POSIX (II)

```
int compteur;
pthread_mutex_t mutc;
pthread_cond_t condc;
void* inc_compt(void *arg)
{
    pthread_mutex_lock(&mutc);
    int temp = compteur;
    compteur = temp+1;
    if (compteur == NB_Thread){
        pthread_cond_signal(&condc)};
    pthread_mutex_unlock(&mutc);
}

void* attend(void *arg)
{
    int continu = 0
    while(continu == 0){
        pthread_mutex_lock(&mutc);
        if (compteur != NB_THREAD)
            {pthread_cond_wait(&condc , &mutc)}
        else {continu = 1};
    }
    pthread_mutex_unlock(&mutc);
}
... <main> ...
```

# Fair Threads: Principe

- ▶ Implémentations (C / Java / Scheme / OCaml) par *Frédéric Boussinot* (inria) dans les années 2000.
  - ▶ <https://www-sop.inria.fr/mimosa/rp/FairThreads/FTC>
  - ▶ s'inscrit dans une série de développements de **programmation réactive**.
- ▶ Idées générales:
  - ▶ **Base**: on s'autorise à demander **explicitement** au programmeur d'indiquer **quand un thread doit rendre la main**.
  - ▶ **ordonnanceurs** (serveurs de synchronisation):
    - ▶ apparaissent **explicitement** dans le langage,
    - ▶ il peut y en avoir **plusieurs**
  - ▶ **liaison**:
    - ▶ chaque thread peut être **lié à au plus un ordonnanceur**.
  - ▶ **ordonnancement**:
    - ▶ les threads liés à **un même ordonnanceur coopèrent** entre eux.
    - ▶ les threads **non-liés**, et les **ordonnanceurs** entre eux se préemptent.

# Fair Threads: Style

- ▶ Avantage des threads systèmes:
  - ▶ chaque **ordonnanceur** et thread **non-lié** peut être exécuté sur un **coeur** différent,
  - ▶ les **entrées/sorties** des threads **non-liés** ne bloquent pas le système
- ▶ Avantage de la coopération:
  - ▶ pris **isolément**, l'exécution d'un **ordonnanceur** et des **threads** qui lui sont liés est **déterministe**
  - ▶ une ressource **partagée au sein d'un ordonnanceur** (et pas en dehors !) n'a pas à être **protégée**
  - ▶ des primitives de communication **intra-ordonnanceur** permettent une manipulation de haut niveau de l'ordonnancement (**attentes**).
- ▶ les ressources **partagées** entre threads **non-liés** (ou entre ordonnanceur) doivent être **protégées**.

# Fair Threads: Caractéristiques

- ▶ **Instants:**
  - ▶ durée interne à l'ordonnanceur pendant laquelle chaque **thread** lié s'exécute jusqu'à une **coopération explicite**.
  - ▶ les threads sont toujours exécutés dans le **même ordre**, à chaque **instant**,
  - ▶ des primitives permettent de **compter les instants**,
  - ▶ héritage de la programmation **réactive**
- ▶ **Évènements:**
  - ▶ mécanisme de **synchronisation** et **communication** intra-ordonnanceur
- ▶ **Automates:**
  - ▶ implémentation ultra-légère de **petits threads**.

# Ordonnancement

- ▶ **Ordonnanceur:**
  - ▶ serveur de **synchronisation**:
    - ▶ à chaque **instant**, effectue une tâche jusque synchronisation explicite pour chaque thread lié.
  - ▶ serveur de **communication**:
    - ▶ permet la **diffusion** d'information à tous les threads liés.
    - ▶ peut être utilisé pour **éveiller** un thread depuis un autre.
  - ▶ serveur d'**exécution**:
    - ▶ lance et maintient des **automates**.
- ▶ l'ordonnanceur est la **brique** de base d'un système FT
  - ▶ réflexion sur l'organisation des **ordonnanceurs**,
  - ▶ un ordonnanceur par **ressource** ?
  - ▶ un ordonnanceur par **type de threads** ? ...

# Ordonnancement coopératif

- ▶ Point de vue de l'**ordonnanceur**:
  - ▶ pendant un **instant**, on élit chaque thread **lié actif**,
  - ▶ on attend, pour chaque **thread**, qui ait rendu explicitement la main (**point** de coopération),
  - ▶ on passe ensuite à l'**instant suivant**.
  - ▶ pas de système de **priorité**.
- ▶ Point de vue du **thread lié t**:
  - ▶ la tâche de *t* sera exécutée jusqu'à l'instruction explicite de **coopération**,
  - ▶ chaque **autre thread** actif fera la même chose (une tâche jusqu'à une coopération) entre la coopération de *t* et sa prochaine élection.
  - ▶ des primitives permettent à un thread de **passer son tour** pour l'instant courant:
    - ▶ attente d'un **événement**,
    - ▶ passage de tour **explicite** (en fait, un évènement).

# Automates

- ▶ **petit thread:**
  - ▶ pas de **pile propre**,
  - ▶ code **séquentiel** (liste d'états),
  - ▶ s'exécute **au sein** d'un ordonnanceur,
- ▶ **change** d'état à chaque instant,
  - ▶ **implicitement**: passe à l'état suivant,
  - ▶ **explicitement**: saut
- ▶ état **final** explicite,
- ▶ accède aux **mécanismes d'évènement**.

# Evènements

- ▶ principe **Abonnement/Diffusion**:
  - ▶ abonnement: chaque thread lié à l'ordonnanceur voit l'évènement,
  - ▶ diffusion: opération **explicite**,
- ▶ mécanisme **d'attente** qui permet de **désactiver** un thread
- ▶ communication par **événements** (passage de **valeur**)
- ▶ utilisation de **tableaux d'évènements** et de mécanisme de **sélection**
  - ▶ **réception alternative**.

# FT en C : Ordonnanceur

- ▶ `#include <fthread.h>`: bibliothèque FT,
- ▶ `ft_scheduler_t`: type d'un ordonnanceur,
- ▶ `ft_thread_t`: type d'un fair thread,
- ▶ `ft_scheduler_t ft_scheduler_create (void)`: création d'un ordonnanceur.
  - ▶ on peut lui attacher des threads avant de le démarrer,
- ▶ `int ft_scheduler_start (ft_scheduler_t sched)`
- ▶ `int ft_scheduler_stop (ft_thread_th)`: force l'arrêt d'un thread,
  - ▶ le thread est effectivement stoppé à la fin de l'instant,
  - ▶ une fonction d'arrêt, passée à la création du thread, est lancée
- ▶ `int ft_scheduler_suspend (ft_thread_th)`: suspend th,
  - ▶ à partir du prochain instant, le thread n'a plus la main,
- ▶ `int ft_scheduler_resume (ft_thread_th)`: reprend l'exécution d'un thread suspendu,
  - ▶ à partir du prochain instant, le thread aura à nouveau la main à chaque instant,
  - ▶ contrôle fin sur les threads.

# FT en C : Threads

- ▶ type `ft_thread_t`,
- ▶ `ft_thread_t ft_thread_create (`  
    `ft_scheduler_t sched ,`  
    `void (*Runnable)(void *),`  
    `void (*cleanup)(void *),`  
    `void *args`  
)
- ▶ un thread est créé avec un `ordonnanceur` auquel il est attaché,
- ▶ `Runnable` est la fonction exécutée par le thread (elle contient des `coopérations explicites`)
- ▶ `cleanup` est la fonction exécutée à l'`arrêt` du thread,
- ▶ `args` arguments passés aux fonctions.

# FT en C : Terminaison

- ▶ Un thread **termine** quand:
  - ▶ il **termine** sa fonction `runnable`,
  - ▶ cette dernière appelle `void ft_exit (void)`,
  - ▶ quelqu'un appelle `ft_scheduler_stop` sur lui,
- ▶ la fonction d'arrêt (`cleanup`) est appelée à l'instant suivant la terminaison.
- ▶ un thread peut **attendre explicitement** la terminaison d'un autre:
  - ▶ `ft_thread_join (ft_thread_t th)` attend la fin d'un thread,
  - ▶ `ft_thread_join_n (ft_thread_t th, int n)` attend la fin d'un thread ou `n` instant (selon ce qui arrive en premier),

# FT en C : Coopération

- ▶ `int ft_thread_cooperate (void): rend la main` à l'ordonnanceur
- ▶ `int ft_thread_cooperate_n (void): rend la main` à l'ordonnanceur et dort pour `n` instants. C'est:  
`for ( i=0;i<k ; i++) ft_thread_cooperate ();`

# FT en C : Exemple

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

void ping (void *id) {
    while (1) {
        fprintf (stderr , "Ping\n");
        ft_thread_cooperate ();
    }
}

void pong (void *id) {
    while (1) {
        fprintf (stderr , "Pong\n");
        ft_thread_cooperate ();
    }
}
```

## FT en C : Exemple (II)

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

int main(void) {
    ft_scheduler_t sched1 = ft_scheduler_create ();
    ft_scheduler_t sched2 = ft_scheduler_create ();
    ft_thread_create (sched1, ping, NULL, NULL);
    ft_thread_create (sched2, pong, NULL, NULL);
    ft_scheduler_start (sched1);
    ft_scheduler_start (sched2);
    ft_exit ();
    return 0;
}
```

# FT en C : Liaison

- ▶ **détacher:** `int ft_thread_unlink (void)`
  - ▶ appelé **depuis** le thread à détacher,
  - ▶ il doit être **lié** à un ordonnanceur,
  - ▶ passe dans l'état **non-lié**.
- ▶ **lier:** `int ft_thread_link (ft_scheduler_t sched)`
  - ▶ appelé **depuis** le thread à détacher,
  - ▶ il doit être **non-lié** à un ordonnanceur,
  - ▶ l'ordonnanceur commence son exécution au **prochain instant**.
- ▶ en utilisant les deux primitives on peut faire **migrer** des threads.

# FT en C : Lecture non-bloquante

```
#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;
}
```

- ▶ le thread **se détache** pour lire un fichier.
- ▶ puis il se relie **au même** ordonnanceur.
- ▶ **ft\_thread\_scheduler** permet de récupérer l'ordonnanceur courant.

# FT en C: 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);
}

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;
}
```

# Implémentation

- ▶ surcouche des threads POSIX
- ▶ 1800 lignes de C
- ▶ ordonnanceur:

```
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;  
};
```

# Implémentation (II)

## ► thread:

```
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;
};
```

## ► instant:

```
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
}
```

# Evènements

- ▶ évènements: communication intra-ordonnanceur
- ▶ type: ft\_event\_t
- ▶ création:
  - ▶ `ft_event_t ft_event_create (ft_scheduler_t sched);`
  - ▶ création séparée de l'activation.
- ▶ activation immédiate:
  - ▶ `int ft_thread_generate (ft_event_t evt);`: l'évènement existe pour l'instant courant uniquement,
  - ▶ équité d'évènement garantie: un thread est réveillé par l'évènement même s'il s'est endormi dans le même instant, avant la génération.
  - ▶ `int ft_thread_generate_value (ft_event_t evt, void *val);`: une valeur est associée à l'évènement
- ▶ activation prochaine:
  - ▶ `int ft_scheduler_broadcast (ft_event_t evt);` l'évènement apparaitra a l'instant suivant.
  - ▶ `int ft_scheduler_broadcast_value (ft_event_t evt, void *val);`

# Evènements (II)

## ► attente:

- ▶ `int ft_thread_await (ft_event_t evt);` suspend l'exécution du thread jusqu'à la génération d'un évènement donné.
- ▶ `int ft_thread_await_n (ft_event_t evt, int n);`: idem, mais attend au plus n instants,

## ► écoute:

- ▶ `int ft_thread_get_value (ft_event_t event,int num,void **result);` récupère la valeur d'un évènement.
  - ▶ si elle existe, elle est stockée et l'appel termine,
  - ▶ sinon la fonction renvoie NULL au prochain instant.

# Sélection

- ▶ la sélection est l'attente de **plusieurs évènements** et le choix d'un **comportement différent** en fonction de l'évènement produit.
  - ▶ **idiomatique** en passage de message,
  - ▶ plus puissant qu'un **case\_of/match** (on peut écouter sur plusieurs canaux différents)
- ▶ `int ft_thread_select(int len, ft_event_t *array, int *mask)`
  - ▶ array et mask ont longueur len,
  - ▶ le thread est suspendu jusqu'à un instant où **au moins un** évènement de array est généré
  - ▶ dans ce cas, les booléens correspondants de mask sont mis à 1 pour chaque évènement généré.
- ▶ `int ft_thread_select_n (int len,ft_event_t *array, int *mask,int timeout)`
  - ▶ comme d'habitude, attente **bornée**

# Exemple (événement)

```
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");
}

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 (événement) (II)

```
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);
}
```

- ▶ sortie standard:

## Exemple (événement) (II)

```
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);
}
```

- ▶ sortie standard: (tout dans le même instant)

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

## Exemple (tableau)

```
ft_event_t a,b;

void awaiteur (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]\n" ,result[0],result[1]);
    if (result[0] == 0 || result[1] == 0) {
        ft_thread_await (result[0]==0 ? events[0] : events[1]);
    }
    fprintf (stdout , "got_both!\n");
    ft_thread_cooperate ();
    fprintf (stdout , "exit!\n");
    exit (0);
}

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

- ▶ attend deux évènements en sélectionnant sur les deux,
- ▶ attend ensuite celui qui reste.
- ▶ trace égrène les instants (classique).

## Exemple (tableau) (II)

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

void bgenerator (void *args)
{
    ft_thread_cooperate_n (3);
    fprintf (stdout , "gen_b_!_");
    ft_thread_generate (b);
}

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;
}
```

## Exemple (tableau) (III)

- ▶ Sortie standard ?:

## Exemple (tableau) (III)

- ▶ Sortie standard ?:

```
instant 1:  
instant 2:  
instant 3:  
instant 4: gen a ! result: [1,0] gen b ! got both!  
instant 5: exit!
```

- ▶ encore une fois, tout se passe dans un même instant,
- ▶ magie du synchrone.

# Automates

- ▶ thread **ultra-léger**,
- ▶ type **ft\_automaton\_t**,
- ▶ création

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

- ▶ défini par des **macros** décrivant **états** et **transitions**

## Exemple (automate) (II)

```
ft_event_t event1, event2;
DEFINE_AUTOMATON (autom)
{
    BEGIN_AUTOMATON
        STATE_AWAIT (0, event1);
        STATE_AWAIT (1, event2)
    { fprintf (stdout, "got_both_!_"); }
    END_AUTOMATON
}

void generator (void *args)
{
    ft_thread_cooperate_n (4);
    fprintf (stdout, "gen_event1_!_");
    ft_thread_generate (event1);
    ft_thread_cooperate_n (4);
    fprintf (stdout, "gen_event1_and_event2_!_");
    ft_thread_generate (event1);
    ft_thread_generate (event2);
    ft_thread_cooperate ();
    fprintf (stdout, "exit\n");
    exit (0);}
```

- ▶ + main idoine
  - ft\_automaton\_create (sched, autom, NULL, NULL)
- ▶ attend que les deux évènements soient générés.

## Exemple (automate) (II)

```
DEFINE_AUTOMATON ( killer )
{
    void **args = ARGS;
    ft_event_t event    = args[0]
    ft_thread_t thread = args[1]

BEGIN_AUTOMATON

    STATE_AWAIT ( 0 , event )
    {
        ft_scheduler_stop ( thread );
    }

END_AUTOMATON
}
```

- ▶ reçoit à la création un évènement et un thread,
- ▶ arrête à l'instant où l'évènement est généré.
- ▶ pas besoin de p\_thread (s'execute dans l'ordonnanceur).

## Exemple (automate) (IV)

```
DEFINE_AUTOMATON (switch_aut)
{
    void **args = ARGS;

    ft_event_t event = args[0]
    ft_thread_t thread1 = args[1]
    ft_thread_t thread2 = args[2]

BEGIN_AUTOMATON

    STATE (0)
    {   ft_scheduler_resume (thread1);}

    STATE_AWAIT (1, event)
    {   ft_scheduler_suspend (thread1);
        ft_scheduler_resume (thread2);
        GOTO(2);}

    STATE_AWAIT (2, event)
    {   ft_scheduler_suspend (thread2);
        ft_scheduler_resume (thread1);
        GOTO(1);}

END_AUTOMATON}
```

- ▶ arbitre entre deux threads,
- ▶ exactement un des deux threads progresse à chaque instant.
- ▶ ne consomme pas les ressources d'un p\_thread\_t.

# Exercice FT de l'examen PC2R de 2016

- ▶ **système:**
  - ▶ 10 **robots** numérotés de 1 à 10, **répartis** sur deux **postes**.
  - ▶ 2 **paniers** un "pair" et un "impair",
  - ▶ 2 **compteurs** de robot (un par poste)
- ▶ **travail** des robots, à chaque **poste**
  - ▶ prendre **un nombre aléatoire**,
  - ▶ **dormir** (les autres robots du poste peuvent continuer de travailler),
  - ▶ incrémenter le **panier** correspondant à la **parité** du nombre,
  - ▶ **dormir**
  - ▶ choisir le poste **le moins chargé** pour recommencer à travailler, et mettre à jour les compteurs.

# Exercice FT de l'examen PC2R de 2016 (II)

```
ft_scheduler_t scheduler_a, scheduler_b;
int nb_robots_a, nb_robots_b;
int panier_pair, panier_impair;

pthread_mutex_t clef_sur_random;
pthread_mutex_t clef_sur_nb_robots_a_et_b;
pthread_mutex_t clef_sur_panier_pair, clef_sur_panier_impair;

void traceinstants (void *arg) {
    int i = 1;

    for (;;) {
        printf (">>>>_instant_%d_du_scheduler_%s.\n", i, (char *)arg);
        fflush(NULL);
        ++i;
        ft_thread_cooperate ();
    }
}
```

# Exercice FT de l'examen PC2R de 2016 (III)

```
void robot (void *arg) {
    int n;

    for (;;) {
        pthread_mutex_lock(&clef_sur_nb_robots_a_et_b);
        if (ft_thread_scheduler() == scheduler_a) {
            —nb_robots_a;
        } else if (ft_thread_scheduler() == scheduler_b) {
            —nb_robots_b;
        }
        pthread_mutex_unlock(&clef_sur_nb_robots_a_et_b);
        ft_thread_unlink();

        pthread_mutex_lock(&clef_sur_random);
        n = rand();
        pthread_mutex_unlock(&clef_sur_random);
        printf ("robot_%d_a_pris_la_valeur_%d.\n", arg, n);
        fflush(NULL);
    }
}
```

# Exercice FT de l'examen PC2R de 2016 (IV)

```
if ((n % 2) == 0) {
    pthread_mutex_lock(&clef_sur_panier_pair);
    printf("robot %d incremente panier_pair : nouvelle valeur = %d\n",
           arg, ++panier_pair);
    fflush(NULL);
    pthread_mutex_unlock(&clef_sur_panier_pair);
} else {
    pthread_mutex_lock(&clef_sur_panier_impair);
    printf("robot %d incremente panier_impair : nouvelle valeur = %d\n",
           arg, ++panier_impair);
    fflush(NULL);
    pthread_mutex_unlock(&clef_sur_panier_impair);
}
usleep(rand() / RAND_MAX * 1000);
```

# Exercice FT de l'examen PC2R de 2016 (V)

```
pthread_mutex_lock(&clef_sur_nb_robots_a_et_b );
if ( nb_robots_a > nb_robots_b ) {
    ++nb_robots_b;
    pthread_mutex_unlock(&clef_sur_nb_robots_a_et_b );
    printf("robot %d va dans le scheduler_b.\n", arg);
    fflush(NULL);
    ft_thread_link(scheduler_b );
} else {
    ++nb_robots_a;
    pthread_mutex_unlock(&clef_sur_nb_robots_a_et_b );
    printf("robot %d va dans le scheduler_a.\n", arg);
    fflush(NULL);
    ft_thread_link(scheduler_a );
}
}
```

- ▶ Modèle de concurrence développé pour *Ocsigen*.
- ▶ Vision **coopérative** de la concurrence
  - ▶ les threads ne sont **pas préemptés** et s'exécutent jusqu'à un `yield` explicite
- ▶ les threads sont des **promesses** de type '`a Lwt.t` qui peuvent être;
  - ▶ en attente `Sleep`, pas encore complétée,
  - ▶ finie `Return x`, complétée avec la valeur `x`,
  - ▶ ratée `Fail exn` (habituellement, à cause d'une IO).
- ▶ **exemple:**

```
# Lwt_io.read_char;;
- : Lwt_io.input_channel -> char Lwt.t = <fun>
```

- ▶ prend en entrée un **canal**,
- ▶ renvoie **une promesse** de caractère.
  - ▶ qui sera **complétée** une fois la `lecture` effectuée.
- ▶ tout s'execute sur un **unique** thread, mais les entrées-sorties sont non-bloquantes.
  - ▶ **coopération** entre les tâches.

## Lwt (II)

- ▶ les promesses s'enchainent dans un style monadique
  - ▶ Lwt.return: 'a -> 'a Lwt.t: crée une promesse déjà complétée,
  - ▶ Lwt.bind : 'a Lwt.t -> ('a -> 'b Lwt.t) -> 'b Lwt.t:
    - ▶ prend une promesse de 'a et une fonction qui prend un 'a et produit une promesse de 'b.
    - ▶ renvoie une promesse de 'b
    - ▶ bind p f attend la compléition de p et passe le résultat à f.
- ▶ c'est bien une monade:
  1. bind(return(x), f) = f(x) (neutralité à gauche)
  2. bind(m, return) = m (neutralité à droite)
  3. bind(bind(m,f),g) = bind(m, x ↦ bind(f(x),g)) (associativité)
- ▶ run essaye de compléter une monade,
- ▶ enchainement naturel:

```
let (>>=) = Lwt.bind;;
let rec f () = Lwt_io.printl "Ping" >>=
  (fun () -> (Lwt_unix.sleep 4.)) >>= f ;;
let rec g () = Lwt_io.printl "Pong" >>=
  (fun () -> (Lwt_unix.sleep 3.)) >>= g ;;
Lwt_main.run (f (), Lwt_main.run (g()));;
```

## Lwt (III)

- ▶ **map** en Lwt:

- ▶ 

```
let rec map f l =
  match l with
  | [] -> Lwt.return []
  | v :: r ->
    let t = f v in
    let rt = map f r in
    t >>= fun v' -> rt >>= fun l' -> Lwt.return (v' :: l')
```

- ▶ de type `('a -> 'b Lwt.t) -> 'a list -> 'b list Lwt.t`
- ▶ **Intérêt** de Lwt: manipulation **confortable** des entrées-sorties et du multi-threading.
- ▶ **Utilisation:** *Ocsigen*, framework web en *OCaml*.

# Conclusion

- ▶ Résumé:
  - ▶ grande **expressivité** du modèle préemptif dans la plupart des langages,
  - ▶ recherche de **techniques** de programmation plus contraintes pour éviter les écueils (FT, Lwt).
  - ▶ les *Fair Threads* sont un peu "legacy", *Lwt* est restreint à *Ocsigen* (i.e. *BeSport*) mais la **coopération** reste un modèle significatif:
    - ▶ boucle d'évènements *Javascript*
    - ▶ multitâche coopératif *Node.js*,
- ▶ TD / TME:
  - ▶ **TD**: préemption + FT
  - ▶ **TME**: programmation FT (Producteurs/Consommateurs)
- ▶ Séance prochaine:
  - ▶ passage de **messages**: canaux synchrones.