Les processus légers : *threads*

Les threads

— Les threads sont des **processus légers** exécutés « à l'intérieur » d'un processus
— L'exécution des threads est concurrente
— Il existe toujours au moins un thread : le thread principal
— La durée de vie d'un thread ne peut pas dépasser celle du processus qui l'a crée
— Les threads d'un même processus *partagent la même mémoire*
— Le coût de création moindre que pour les **processus lourds** ; échange de (grosses) structures de données sans recopie, par échange de pointeur.
— Mais cela est risqué ! Controler la synchronisation, en utilisant des verrous.

Threads préemptifs vs Threads coopératifs

— Threads préemptifs
  — Le système peut suspendre l'exécution d'un thread à tout moment pour exécuter un autre thread
  — Problème du non-déterminisme
— Threads coopératifs
  — Chaque thread décide quand il peut être suspendu
  — Problème de réactivité (un thread ne rend jamais la main !)

— Dans ce cours : threads préemptifs
— Il existe des librairies de threads coopératifs qui miment les threads Posix. E.g., `Lwt` et `Async` en OCaml.
— Certains langages s'appuient sur un modèle de parallélisme coopératif auquel ils ajoutent des structures de controle (suspension/interruption) en garantissant le déterminisme et l'absence de blocage (e.g., ReactiveML).

Les threads Posix : *Pthread*

- La création
  - `#include <pthread.h>`
    `int pthread_create(pthread_t *pthread,`
    `const pthread_attr_t *attr,`
    `void *(*fonction)(void *), void *arg);`
  - `thread` : pour récupérer l'identité du thread qui est créé
  - `attr` : attributs du thread.
    Si `attr` est `NULL`, la valeur par défaut est prise
  - `fonction` : fonction à exécuter en parallèle
  - `arg` : arguments de la fonction
  - en cas de succès, renvoit `NULL` ; code d'erreur sinon.
- Identité d'un thread
  - `pthread_t pthread_self(void);`
- Égalité ente threads
  - `int pthread_equal(pthread_t t1, pthread_t t2);`

Dans linux, ils sont implémentés dans la librairie `libc`.

Voir le code source ici :

`https://sourceware.org/git/?p=glibc.git;a=tree;f=nptl;`
`h=d0ce23d37e9b77d6ff82ffdee0f7a1f8f137aa41;hb=HEAD`

Implémentation des pthreads :
`http://maxim.int.ru/bookshelf/PthreadsProgram/htm/r_47.html`

Linux : `http://www.evanjones.ca/software/threading.html`

En Linux, un thread et un processus lourd se comportent de la même manière :

- plusieurs threads peuvent être exécutés en parallèle sur des coeurs différents (comme les processus) ;
- Deux threads d'un même processus partagent la même zone mémoire, donc la table des pages.

Exemple : gcc thread-0.c -lpthread (thread-0.c)

```
void *f(void *arg) {
  int boucle;
  while (1) {
    printf("Je suis la Pthread\n");
    for(boucle=0; boucle < 10000000; boucle++);
  }
  return NULL;
}
int main() {
  pthread_t tid;
  int rep, boucle;
  printf("Processus %d lance une Pthread\n", getpid());
  if ( 0 == (rep = pthread_create(&tid, NULL, f, NULL)) ) {
    printf("Pthread creee\n");
  } else {
    perror("pthread_create");
  }
  while (1) {
    printf("Je suis la fonction main\n");
    for(boucle=0; boucle < 10000000; boucle++);
  }
  return 0;
}
```

Exemple : gcc thread-1.c -lpthread (thread-1.c)

```
#include <stdio.h>
#include <pthread.h>
pthread_t tid[3];
char *nom[3] = { "ainee", "cadette", "benjamine" };
void *annexe(void *arg) {
  int boucle;
  printf("Je suis la Pthread %s d'identite %u\n",
         (char *)arg, (int)pthread_self());
  for(boucle=0; boucle < 10000000; boucle++);
  printf("%s: je suis apres la premiere boucle\n", (char *)arg);
  for(boucle=0; boucle < 10000000; boucle++);
  printf("%s: je suis apres la deuxieme boucle\n", (char *)arg);
  return NULL; }
int main() {
  int ind, rep;
  printf("Processus %d lance\n", getpid());
  for (ind=0; ind<3; ind++) {
    if ( 0 == (rep = pthread_create(tid+ind, NULL, annexe, nom[ind])) ) {
      printf("Pthread %d creee\n", ind);
    } else { fprintf(stderr, "%d : ", ind); perror("pthread_create"); }
  }
  sleep(20); return 0; }
```

Exemple (thread-2.c)

— durée de vie limitée à celle du processus

```
#include <stdio.h>
#include <pthread.h>
void *immortelle(void *arg) {
  while (1) { printf("coucou\n"); }
}
int main() {
  pthread_t tid;
  int rep;
  if ( 0 == (rep = pthread_create(&tid, NULL, immortelle, NULL)) ) {
      printf("Pthread creee\n");
    } else { perror("pthread_create"); }
  sleep(1);
  return 0;
}
```

Terminaison (thread-4.c)

```
void *f(void *arg) {
  exit(1);
}
int main() {
  pthread_t tid;
  int rep;
  if ( 0 == (rep = pthread_create(&tid, NULL, f, NULL)) ) {
    printf("Pthread creee\n");
  } else { perror("pthread_create"); }
  sleep(5);
  printf("FIN");
  return 0;
}
```

— Le message `FIN` est-il affiché ?

Terminaison

— Terminaison d'un thread
  — `void pthread_exit(void *value_ptr);`
    — Si le thread a l'attribut `PTHREAD_CREATE_JOINABLE`, à sa mort, le thread devient un « zombie » jusqu'à ce qu'un autre thread en prenne connaissance
    — Si le thread a l'attribut `PTHREAD_CREATE_DETACHED`, à sa mort, le thread disparaît directement
— L'attribut `detachstate`
  — il peut être fixé à la création du thread
  — être modifié : `int pthread_detach(pthread_t thread);`

Terminaison (thread-5.c)

```
void *f(void *arg) {
  pthread_exit(NULL);
}
int main() {
  pthread_t tid;
  int rep;
  if ( (rep = pthread_create(&tid, NULL, f, NULL)) == 0) {
    printf("Pthread creee\n");
  } else { perror("pthread_create"); }
  sleep(5);
  printf("FIN");
  return 0;
 }
```

Synchronisation

— Attendre la fin d'un thread
  — `int pthread_join(pthread_t tid, void **value)`
  — Appel bloquant jusqu'à ce que le thread `tid` termine (ou soit résilié, cf `pthread_cancel`)
  — Si `value` n'est pas `NULL` pointeur vers la valeur de retour du thread (ou, pointeur vers `PTHREAD_CANCELED` en cas de résiliation)
  — `tid` ne doit pas être détaché
  — `pthread_join` revoie `0` en cas de succès (`errno` n'est pas positionné en cas d'échec)

Exemple (thread-6.c)

```
pthread_t tid[3];
void *annexe(void *arg) {
  int boucle;
  printf("Je suis la Pthread %d d'identite %d\n",
         *((int *)arg), (int)pthread_self());
  *((int *)arg) =  (*((int *)arg) + 1);
  pthread_exit(arg); }
int main() {
  int ind, rep;
  int *valeur;
  printf("Processus %d lance\n", getpid());
  for (ind=0; ind<3; ind++) {
    int *pi = malloc(sizeof(int)); *pi=ind;
    if ( 0 == (rep = pthread_create(tid+ind, NULL, annexe, pi)) ) {
      printf("Pthread %d creee\n", ind);
    } else { fprintf(stderr, "%d : ", ind); perror("pthread_create"); }
  }
  for (ind=0; ind<3; ind++) {
    pthread_join(tid[ind], (void **) &valeur);
    printf("Fin de la Pthread d'identite %d et de valeur %d\n",
           (int)tid[ind], *valeur);
  } return 0; }
```

Mémoire partagée (conc-1.c)

```
int cpt = 0;
void *incr(void *arg) {
  int i;
  printf("[%d]: adr de i = %p, adr de cpt = %p\n", (int)pthread_self(), &i, &cpt);
  for(i=0; i < 1000000; i++) {
    if ( i % 10000 == 0) { printf("[%d]: i = %d\n", (int)pthread_self(), i); }
    cpt++;
  }
  printf("[%d]: FIN\n", (int)pthread_self());
  return NULL;
}
int main() {
  pthread_t tid[2];
  pthread_create(tid, NULL, incr, NULL);
  pthread_create(tid+1, NULL, incr, NULL);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("cpt = %d", cpt);
}
```

Sections critiques

— Les accès en lecture et en écriture à des données partagées doivent être limités à un seul thread à la fois :
  — les threads doivent être en *exclusion mutuelle*
— Les parties du code qui doivent être accédées par au plus un processus sont des *sections critiques*
— Pour que des threads coopèrent correctement et efficacement, ils doivent respecter les conditions suivantes :

1. Deux processus ne peuvent pas être simultanément dans une section critique relative à une ressource commune
2. Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs
3. Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres
4. Aucun processus ne doit attendre trop longtemps avant de pouvoir entrer dans sa section critique

Exclusion mutuelle par accès atomiques en mémoire

- La programmation de l'exclusion mutuelle est difficile (au moyen de la seule indivisibilité des accès en mémoire)
- Plusieurs solutions erronées ont été publiées
  - Pour montrer qu'une solution est fausse, on construit un scénario avec des entrelacements d'opérations qui conduisent à un problème
- Les preuves de corrections sont difficiles
  - Première solution prouvée correcte publiée par Deckker et Dijkstra en 1966
  - Peterson à publié une solution plus simple en 1981
- Solutions reposant sur de l'attente active
  - test répétitifs des variables utilisées pour la synchronisation

Solution 1 (fausse)

— Variables partagées : `isc1`, `isc2` (vraies si intention d'entrer en section critique)

— Initialement : `isc1 = FAUX`, `isc2 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  isc1 = VRAI;`
`  while (isc2) { ; }`
`  // section critique ...`
`  isc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  isc2 = VRAI;`
	`  while (isc1) { ; }`
	`  // section critique ...`
	`  isc2 = FAUX;`
	`  // section restante`
	`done`

— Scénario fautif ?

Solution 1 (fausse)

— Variables partagées : `isc1`, `isc2` (vraies si intention d'entrer en section critique)

— Initialement : `isc1 = FAUX`, `isc2 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  isc1 = VRAI;`
`  while (isc2) { ; }`
`  // section critique ...`
`  isc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  isc2 = VRAI;`
	`  while (isc1) { ; }`
	`  // section critique ...`
	`  isc2 = FAUX;`
	`  // section restante`
	`done`

— Scénario fautif : P1 affecte `isc1 = VRAI`; P2 affecte `isc1 = VRAI`;

— $\Rightarrow$ interblocage

Solution 2 (fausse)

— Variables partagées : `sc1`, `sc2` (vraies si en section critique)

— Initialement : `sc1 = FAUX`, `sc2 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  while (sc2) { ; }`
`  sc1 = VRAI;`
`  // section critique ...`
`  sc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (sc1) { ; }`
	`  sc2 = VRAI;`
	`  // section critique ...`
	`  sc2 = FAUX;`
	`  // section restante`
	`done`

— Scénario fautif ?

Solution 2 (fausse)

— Variables partagées : `sc1`, `sc2` (vraies si en section critique)

— Initialement : `sc1 = FAUX`, `sc2 = FAUX`

Processus 1	Processus 2

`while (VRAI) {`
`  while (sc2) { ; }`
`  sc1 = VRAI;`
`  // section critique ...`
`  sc1 = FAUX;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (sc1) { ; }`
	`  sc2 = VRAI;`
	`  // section critique ...`
	`  sc2 = FAUX;`
	`  // section restante`
	`done`

— Scénario fautif :

  — P1 test `sc2` à faux ; P2 test `sc1` à faux

  — P1 affecte `sc1 = VRAI` ; P2 affecte `sc1 = VRAI` ;

  — P1 et P2 sont en même temps en section critique !

Solution 3 (fausse)

— Variables partagées : `tour`

— Initialement : `tour = 1`

Processus 1	Processus 2

`while (VRAI) {`
`  while (tour != 1) { ; }`
`  // section critique ...`
`  tour = 2;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (tour != 2) { ; }`
	`  // section critique ...`
	`  tour = 1;`
	`  // section restante`
	`done`

— Scénario fautif ?

Solution 3 (fausse)

— Variables partagées : `tour`

— Initialement : `tour = 1`

Processus 1	Processus 2

`while (VRAI) {`
`  while (tour != 1) { ; }`
`  // section critique ...`
`  tour = 2;`
`  // section restante`
`done`
	`while (VRAI) {`
	`  while (tour != 2) { ; }`
	`  // section critique ...`
	`  tour = 1;`
	`  // section restante`
	`done`

— Scénario fautif : P1 stoppé hors section critique

— $\Rightarrow$ P2 ne peut pas entrer en section critique

Solution de Peterson (peterson.c)

```
#define N 2
#define VRAI 1
#define FAUX 0
int tour; /* a qui le tour */
int interesse[N] = { FAUX, FAUX }; /* initialise a FAUX */

void entrer_region(int process) {
  int autre = 1-process;
  interesse[process] = VRAI;
  tour = autre;
  while ( (interesse[autre] == VRAI) && (tour == autre) ) { ; }
}
void quitter_region(int process) {
  interesse[process] = FAUX;
}
```

Cet algorithme est juste sur un processeur mais faux sur les machines multi-coeur modernes ayant un modèle mémoire faible [a].

a. Lire ceci :https://www.mjmwired.net/kernel/Documentation/memory-barriers.txt#305

Interlude : comment prouver Peterson ?

Cas de deux processus : $P_1||P_2$ en parallèle. Montrer que les deux ne sont jamais dans la section critique in_i, où $i \in \{1, 2\}$. :

```
P1 = while true do
       out: y1 := true; tour := 2
       req: wait (not y2) or (tour = 1)
       in: <section critique>
       exit: y1 := false

P2 = while true do
       out: y2 := true; tour := 1
       req: wait (not y1) or (tour = 2)
       in: <section critique>
       exit: y2 := false
```

P_1 et P_2 peuvent être représentés sous la forme d'un système de transitions.

Système de transitions

Représenter chaque programme sous la forme d'un **système de transitions**. La composition parallèle est le produit.

Système de transition : $M = (S_0, S, R)$ défini par :

- S_0 : ensemble (fini) d'états initiaux ($S_0 \subseteq S$) ;
- S : ensemble (fini) d'états ;
- R : relation de transition totale (pour tout s, il existe t tq $R(s,t)$).

Un chemin de M de début s_0 est une suite $s_0, s_1, ..., s_k$ tq $R(s_i, s_{i+1})$ pour tout $i \geq 0$. Le chemin est initialisé si $s_0 \in S_0$.

Système de transition étiqueté : $M = (S_0, S, E, R)$.

- E est un ensemble d'étiquettes.
- $R \in S \times E \times S$ est une relation totale : pour tout $q \in S$, il existe $e \in E$ et $q' \in S$ tel que $(q, e, q') \in R$.
- Notation : $q \xrightarrow{e} q'$ lorsque $(q, e, q') \in R$.

- Représentation explicite = un graphe ;
- représentation implicite = une formule booléenne entre variables et variables d'état.

Représentation implicite

Représentation implicite : $M = (S_0, V, R)$.

— V est un ensemble fini de symboles de variables. Chaque variable $x \in V$ prend ses valeurs dans un domaine D_x fini (e.g., booléen, type énuméré)

— S_0 est une assertion (formule booléenne) sur V. Ex : $(x = 0) \wedge (y = 1)$.

— R est une assertion sur $V \times V'$. Lorsque $V = \{x_1, ..., x_d\}$, V' dénote $V' = \{x'_1, ..., x'_d\}$, i.e., la copie primée des variables de V.
Ex :

$$(x = 0) \longrightarrow (x' = 1) \qquad (x = 0) \vee (y = 0) \longrightarrow (x' = 0) \vee (y' = 1)$$

Convention : si x ne change pas, implicitement $x' = x$.

Application a Peterson (pour $N = 2$)

Variables : $pc_1, pc_2 : \{out, req, in, exit\}$; $y_1, y_2 : \{false, true\}$; $tour : \{1, 2\}$.

S : valuation des variables ;
$S_0 = (pc_1 = out) \wedge (pc_2 = out) \wedge (y_1 = false) \wedge (y_2 = false)$.

$R(pc_1, pc_2, y_1, y_2, tour)$ est la conjonction de :

$$(pc_1 = out) \longrightarrow (pc_1' = req) \wedge (tour' = 2) \wedge y_1'$$
$$(pc_1 = req) \wedge \neg y_2 \longrightarrow (pc_1' = in)$$
$$(pc_1 = req) \wedge (tour = 1) \longrightarrow (pc_1' = in)$$
$$(pc_1 = in) \longrightarrow (pc_1' = exit)$$
$$(pc_1 = exit) \longrightarrow (pc_1' = out) \wedge \neg y_1'$$
$$(pc_2 = out) \longrightarrow (pc_2' = req) \wedge (tour' = 1) \wedge y_2'$$
$$(pc_2 = req) \wedge \neg y_1 \longrightarrow (pc_2' = in)$$
$$(pc_2 = req) \wedge (tour = 2) \longrightarrow (pc_2' = in)$$
$$(pc_2 = in) \longrightarrow (pc_2' = exit)$$
$$(pc_2 = exit) \longrightarrow (pc_2' = out) \wedge \neg y_2'$$

Preuve automatique de l'algo. de Peterson

Problème à résoudre : la configuration $(pc_1 = in) \wedge (pc_2 = in)$ est-elle accessible ?

L'ensemble des configurations (états) du système étant fini, on peut vérifier cette propriété par énumération. Il suffit ici de vérifier que la propriété est vraie dans chaque état accessible.

C'est le principe du **Model checking** inventé pas Sifakis, Clarke et Emerson qui leur a valu le Prix Turing en 2007.

Dans le cas présent, le système se modélise et se prouve aisément avec l'outil Cubicle (`http://cubicle.lri.fr`) ou
Z3-PDR (`http://rise4fun.com/z3/tutorialcontent/fixedpoints`).

Nous étudierons ces questions dans la deuxième partie du cours.

Fin de l'interlude

Attente passive : les mutex

— L'attente active n'est jamais la bonne solution
  — on préfère endormir un thread plutôt que faire de l'attente active
  — on devra alors être capable de le réveiller
— Les mutex
  — un mutex peut être libre ou verrouillé
  — un thread peut obtenir le verrouillage d'un mutex et en devenir propriétaire
  — un mutex ne peut être verrouillé que par un seul thread à la fois
  — toute demande de verrouillage d'un mutex déjà verrouillé entraîne le blocage du thread qui fait la demande ou l'échec de la demande

Les mutex

— Un mutex est une variable de type `pthread_mutex_t`

— Initialisation d'un mutex

  — `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`

— Verrouillage d'un mutex

  — prendre un verrou existant ; exécution bloquée jusqu'à ce que le verrou soit libéré.

  ```
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  ```

  — (tenter de) prendre le verrou mais ne pas rester bloqué. Retourne 0 en cas de succès (le verrou était libre).

  ```
  int pthread_mutex_trylock(pthread_mutex_t *mutex);
  ```

— Déverrouillage d'un mutex

  — `int pthread_mutex_unlock(pthread_mutex_t *mutex);`

Cf. Fonctions `Thread.create`, `Thread.lock`, `Thread.try_lock`, `Thread.unlock` en OCaml.

Exemple (mutex.c)

```
int cpt = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void *incr(void *arg) {
  int i;
  printf("[%d]: adr de i = %p, adr de cpt = %p\n", (int)pthread_self(), &i, &cpt);
  for(i=0; i < 1000000; i++) {
    if ( i % 10000 == 0) { printf("[%d]: i = %d\n", (int)pthread_self(), i); }
    pthread_mutex_lock(&mutex);
    cpt++;
    pthread_mutex_unlock(&mutex);
  }
  printf("[%d]: FIN\n", (int)pthread_self());
  return NULL;
}
int main() {
  pthread_t tid[2];
  pthread_create(tid, NULL, incr, NULL);
  pthread_create(tid+1, NULL, incr, NULL);
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("cpt = %d", cpt);
  return 0; }
```

Implémentation d'un Mutex

Un mutex peut s'implémenter à partir d'une instruction machine "atomique" `test-and-set`.

"atomique" signifie que lorsqu'un CPU exécute l'instruction, celle-ci ne peut être interrompue en cours d'exécution par un autre CPU.

L'instruction `test-and-set(boolean *lock)` essaie d'écrire la valeur 1 à l'adresse `lock` de manière atomique. Si d'autre processus essaient d'écrire en même temps, ils ne peuvent le faire tant que le premier n'a pas terminé.

La sémantique correspond à la séquence d'instruction exécutée de manière atomique :

```
int test_and_set (int *lock){
     int old = *lock;
     *lock = 1;
     return old;
     }
```

qui s'utilise ainsi :

```
int lock(int *lock)
{
    while (test_and_set(lock) == 1);
}
```

Cf. `http://irl.cs.ucla.edu/~yingdi/web/paperreading/smp_locking.pdf`

Cf. Implementing locks :
`http://web.stanford.edu/class/cs140/cgi-bin/lecture.php?topic=lockImpl`