Threads et atomiques avec C11

Jens Gustedt

INRIA

ICube, Université de Strasbourg

2015

Outline

Table of Contents

C et les standards ISO

la normalisation garantie la portabilité

- mapper un language de haut niveau sur n'importe quel architecture
- insistez que votre compilateur suit le dernier standard !

C[']est une norme ISO

- annés 70, créé par Kernighan et Richie
- C89, première normalisation, le grand succès
- C95, bugfix
- C99, C moderne, difficilement accueillit
- C11

C et les standards ISO

C99

- tableaux avec tailles dynamiques
- spécification formelle du préprocesseur
- objets temporaires (compound literals)
- arithmétique complexe
- `<tgmath.h>`, fonctions mathématiques génériques

C11

- threads et ses structures de contrôle
- types atomiques
- fonctions génériques
- plusieurs ajustements mineurs

Table of Contents

Threads, definitions

Définition d'un thread

- processus léger :
  - plusieurs threads se partagent un *même espace mémoire*
- API C11 : `threads.h`
  - pas encore implanté par tous : `p99_thread.h`
  - utiliser une lib C alternative, musl
    `http://www.musl-libc.org/`

Comparaison avec processus lourds

- pas de séparation d'état entre threads
- plus rapide à lancer
- il n'y a pas de hiérarchie entre threads, tous sont égaux (sauf un)
- `exit` et `return` de `main` termine *tous* les threads d'un processus

Threads, interfaces

Gestion des threads C11

```
/* Thread start and stop */
typedef int (*thrd_start_t)(void*);
int thrd_create(thrd_t *, thrd_start_t, void *);
void thrd_exit (int);

/* Thread information */
thrd_t thrd_current(void);
int thrd_equal(thrd_t, thrd_t);

/* Thread coordination */
int thrd_join(thrd_t, int*);
int thrd_detach(thrd_t);
```

Threads, interfaces

Structures de contrôle threads C11

type		aussi
`thrd_t`	identification de thread	
`tss_t`	thread specific storage	`_Thread_local`
`mtx_t`	exclusion mutuelle	
`cnd_t`	variables de condition	
`once_flag`	initialisation unique	
`atomic_flag`	échange sans attente	`_Atomic`

Threads, exemple de base

fonction exécutée par chaque thread

```
/* User type pour structuré les infos pass'es aux threads */
typedef struct thrd_info thrd_info;
struct thrd_info {
  unsigned num;
  double val;
};
int myThreadFunction(void* arg) {
  /* Interprétation de arg en type voulu, no cast ! */
  thrd_info* info = arg;
  printf("I am %u with value %g\n", info->num, info->val);
  /* retour d'un 'etat `a qqn qui s'int'eresse */
  return info->val > 1.0;
}
```

Threads, exemple de base

```
int main(int argc, char *argv[ ]) {
  unsigned const n = argc-1;
  thrd_info info[n];
  thrd_t id[n];
  for (unsigned i = 0; i < n; ++i) {
     info[i] = (thrd_info){ .num = i, .val = strtod(argv[i+1], 0) };
     thrd_create(&id[i], myThreadFunction, &info[i]);
  }
  for (unsigned i = 0; i < n; ++i) {
    int ret = 0;
    thrd_join(id[i], &ret);
    printf("thread %u returned %d\n", i, ret);
  }
  return EXIT_SUCCESS;
}
```

Threads, exemple de threads détachés

```c
/* cette version a une fuite, manque free(info) */
static thrd_info * info = 0;
int main(int argc, char *argv[ ]) {
  unsigned const n = argc-1;
  /* doit être alloué sur le tas */
  info = malloc(sizeof(thrd_info[n]));
  for (unsigned i = 0; i < n; ++i) {
     thrd_t id;
     info[i] = (thrd_info){ .num = i, .val = strtod(argv[i+1], 0) };
     thrd_create(&id, myThreadFunction, &info[i]);
     thrd_detach(id);
  }
  /* return de main ferait terminer tous le process */
  thrd_exit(EXIT_SUCCESS);
}
```

Threads, exemple de threads détachés

```c
/* sans fuite de mémoire */
static thrd_info * info = 0;
static void cleanup(void) {
  puts("terminating execution");
  free(info);
}
int main(int argc, char *argv[ ]) {
  atexit(cleanup);
  unsigned const n = argc-1;
  /* doit ^etre alloué sur le tas */
  info = malloc(sizeof(thrd_info[n]));
  ...
  /* return de main ferait terminer tous le process */
  thrd_exit(EXIT_SUCCESS);
}
```

Threads, problèmes de cohérence

Le problème de base : la variable critique

```
++a;        /* plusieurs instructions ? */
a += 1;     /* plusieurs instructions ? */
a = a + 1; /* plusieurs instructions ! */
```

en amd64 avec une var locale, si chance :

```
addl    $1, -4(%rbp)
```

pas de chance :

```
movl    a(%rip), %eax
addl    $1, %eax            /* interruption ? */
movl    %eax, a(%rip)       /* écrasement  ! */
```

Threads, problèmes de cohérence

« thread safety »

- *dangereux*:
  - variables globales
  - variables locales `static`
- `libc`
  - `ctime`, `asctime` :
    ne sont pas thread-safe car elles retournent un état statique
  - `printf` et similaire *sont* thread-safe.

read the manual !

Table of Contents

problèmes de base

Comment garantir la cohérence en cas de threads ?

Problèmes :

- multiples instructions assembler pour une instruction C
- multiples étapes pipeline pour une instruction assembler

délais de visibilité de modifications d'objets inter thread

modification thread 1 $\longrightarrow$ mémoire $\longrightarrow$ visibilité thread 2

qqs centaines de cycles

probabilité d'interruption

- est non-nulle
- execution de parties d'instructions dans un ordre quelconque

Atomic, le noveau API C11

comme qualificateur

- syntaxe : `_Atomic toto = ATOMIC_VAR_INIT(007);`
- implanté à partir : `gcc` v. 4.9, `clang` v. 3.5
- chargement et écriture : évaluation `()`, affectation `=`
- arithmetique : opérateurs classiques `++`, `--`, `+=` ...

comme spécificateur

- syntaxe : `_Atomic(unsigned) toto = ATOMIC_VAR_INIT(007);`
- support compilateur peut être émulé : `p99.gforge.inria.fr`
- chargement et écriture : `atomic_load`, `atomic_store`
- arithmétique : `atomic_fetch_add`
- échange conditionnel : `atomic_compare_exchange_weak`

ordonnoncement mémoire

cohérence séquentielle

- Les atomics comme vues ici ont une *cohérence sequentielle*.
- « as-if » un ordre total entre threads sur les opérations atomics.
- sur la plupart des architectures $\approx$ synchronization des caches
- trés couteux en termes de cycles horloge (*centaines*)

à utiliser avec modération

autre modèles de cohérence

- `memory_order` peut être relâché pour chaque opération
- `relaxed` : seul garantie est l'intégralité
- `acquire/release` : pour éviter la synchro complète

Atomic, le noveau API C11

utilisation dans la fonction des threads

```
#include <stdatomic.h>

unsigned volatile _Atomic count = ATOMIC_VAR_INIT(0);

int myThreadFunction(void* arg) {
  /* Interpr'etation de arg en type voulu */
  thrd_info* info = arg;
  unsigned here = count++;
  printf("I am %u with counter %u, now %u\n", info->num, here, count);
  return info->num == here;
}
```

Atomic, le noveau API C11

principales fonctions / macros

```
C atomic_load(volatile A *obj);
void atomic_store(volatile A *obj, C des);
C atomic_exchange(volatile A *obj, C des);
/************************* échanges conditionels ************************/
_Bool atomic_compare_exchange_strong(volatile A *obj, C *attendu, C des);
_Bool atomic_compare_exchange_weak(volatile A *obj, C *attendu, C des);
/************************** arithmetique ********************************/
C atomic_fetch_add(volatile A *obj, M oper);
C atomic_fetch_sub(volatile A *obj, M oper);
/************************** opération de bits ***************************/
C atomic_fetch_and(volatile A *obj, M oper);
C atomic_fetch_or(volatile A *obj, M oper);
C atomic_fetch_xor(volatile A *obj, M oper);
```

Atomic, flags atomiques

un type simple pour la gestion inter-threads : `atomic_flag`

- que deux interfaces de fonction, pas d'opérateurs
- deux états : *clear* et *set*
- l'état n'est que observable en le modifiant

```
/* initialise à l'état "clear" */
atomic_flag flag = ATOMIC_FLAG_INIT;

/* assure l'état "set" et retourne la valeur précédente  */
bool atomic_flag_test_and_set(volatile atomic_flag *);

/* inconditionnellement à l'état "clear" */
void atomic_flag_clear(volatile atomic_flag *);
```

Table of Contents

Section critique

variable critique

- une variable qui
  - peut être modifiée par au moins un thread
  - peut être lue par plusieurs threads

Ce sont souvent des variables globales, ou passées en paramètre à la création du thread.

section critique (SC)

- une partie d'un programme parallèle qui
  - ne doit pas être exécutée par plus qu'un seul thread à la fois

Ce sont souvent des parties de code qui accèdent aux variables critiques.
Si congestion : attente imposée à l'entrée dans la SC

SC avec peu de support matériel

- Algo de Peterson pour thread P_{id} (limité à 2 thread, $id = 0, 1$)

```
static bool volatile inside[2] = { false, false, };
static bool volatile prio = false;
```

```
inside[id] = true;
prio = !id;
do { /* busy wait */ } while (inside[!id] && prio == !id);
/* section critique */
inside[id] = false;
```

Attention:

- `volatile`, sinon les lectures au `while` peuvent être éliminés
- suppose une l'écriture atomic de valeurs Booléens
- suppose une visibilité de l'écriture "immédiate"
- sans utilisation d'atomic, ce n'est pas portable

SC re-écrit avec _Atomic

```
static _Atomic(bool) volatile inside[2] = {
   ATOMIC_VAR_INIT(false), ATOMIC_VAR_INIT(false),
};
static _Atomic(bool) volatile prio = ATOMIC_VAR_INIT(false);
```

```
inside[id] = true;
prio = !id;
do { /* busy wait */ } while (inside[!id] && prio == !id);
/* section critique */
inside[id] = false;
```

- *Remarque :* `volatile` est toujours nécessaire

SC avec atomic_flag

```
static atomic_flag flag = ATOMIC_FLAG_INIT;
```

```
do { /* busy wait */ } while (atomic_flag_test_and_set(&flag));
/* section critique */
atomic_flag_clear(&flag);
```

- Avantages :
  - facile à implanter (`flag` est local, initialisation statique)
  - fonctionne pour plus que 2 threads
  - rapide en situation de non-congestion
  - adapté si la SC est quelques lignes / instructions
- Attention : c'est une attente active
- *Remarque* : `volatile` n'est pas nécessaire, ici, car appel à fonction.

Attente inactive

attente inconditionnel : mutex

```
int mtx_init(mtx_t *, int);
int mtx_destroy(mtx_t *);
```

```
int mtx_lock(mtx_t *);
int mtx_trylock(mtx_t *);
int mtx_unlock(mtx_t *);
```

```
struct timespec { time_t tv_sec; long tv_nsec; };
int mtx_timedlock(mtx_t *, struct timespec *habs);
```

Attente inactive

SC avec mtx_t

```
/* Hors fonction, visible a tous thread */
mtx_t mutex;
```

```
/* interieur de main avant tous threads */
mtx_init(&mutex, mtx_plain);
```

```
/* interieur des threads, attente inactive */
mtx_lock(&mutex);
/* section critique */
mtx_unlock(&mutex);
```

Attente inactive

SC avec `mtx_t`

- Avantages:
  - adapté si la SC est longue et/ou beaucoup de congestion
  - c'est une attente inactive
- Désavantages:
  - initialization dynamique (mais v. POSIX)
  - changement de contexte $\Longrightarrow$ noyau

Attente inactive

Propriétés des mutex

Le deuxième paramètre de `mtx_init` est une combinaison de

```
enum { mtx_plain, mtx_recursive, mtx_timed, };
```

pour définir différents propriétés des mutex.

Mutex "timed"

- `mtx_timed` : le mutex permet l'appel de la fonction

```
struct timespec { time_t tv_sec; long tv_nsec; };
int mtx_timedlock(mtx_t *, struct timespec *habs);
```

Ceci permet de regagner contrôle après le temps demandé, même si le verrou n'est pas acquit.

Attente inactive

Mutex récursif

- `mtx_recursive` se combine avec `mtx_plain` ou `mtx_timed` :
  `mtx_plain | mtx_recursive`
  `mtx_timed | mtx_recursive`
- sans `mtx_recursive` : ne peuvent pas être verrouilles plusieurs fois par le même thread sans `mtx_unlock`
- `mtx_recursive` : autant d'appels `mtx_lock` et `mtx_unlock` just à ce que le verrou est libéré par le thread

Attente inactive

attente conditionnelle avec mutex

```
mtx_lock(&mutex);
while (blabla != 42) {
  mtx_unlock(&mutex);
  /* un autre thread assure la condition */
  mtx_lock(&mutex);
}
/* section critique avec garantie sur la condition */
mtx_unlock(&mutex);
```

doublement mauvais

- c'est encore une attente active
- plein de switch de contexte entre espace utilisateur et noyaud

Attente inactive

attente conditionnelle : conditions

```
int cnd_init(cnd_t *);
void cnd_destroy(cnd_t *);
```

```
int cnd_wait(cnd_t *, mtx_t *);
int cnd_timedwait(cnd_t *, mtx_t *, struct timespec const *);
```

```
int cnd_signal(cnd_t *cond);
int cnd_broadcast(cnd_t *cond);
```

SC avec mtx_t et cnd_t

Hors fonction, visible a tous thread

```
mtx_t mutex;
cnd_t cond;
```

interieur de main avant tous threads

```
mtx_init(&mutex, mtx_plain);
cnd_init(&cond);
```

SC avec `mtx_t` et `cnd_t`

attente inactive : `cnd_wait`

- liaison temporaire entre `cond` et `mutex`
- pendant l'attente `mutex` est relaché
- en revenant le `mutex` est acquis *de nouveau*

Attention : spurious wakeups

`cnd_wait` doit toujours être

- dans une boucle qui
- teste la vrai condition

SC avec mtx_t et cnd_t

interieur d'un des threads, attente inactive

```
mtx_lock(&mutex);
/* test de la condition et blocage possible */
while (blabla != 42) cnd_wait(&cond, &mutex);
/* section critique avec garantie sur la condition */
mtx_unlock(&mutex);
```

interieur d'un autre thread, signalisation

```
mtx_lock(&mutex);
/* assure la condition */
blabla = 42;
/* reveille tous les threads en attente */
cnd_broadcast(&cond);
mtx_unlock(&mutex);
```

Table of Contents

Threads, séparation des états

variables local d'un thread

- le plus important: garder les variables en local (auto)
- si ce n'est pas possible
  - `_Thread_local` (C11)
  - `__thread` (gcc)
  - ou *thread-specific storage*, TSS

```
typedef void (*tss_dtor_t)(void*);
int tss_create(tss_t *key, tss_dtor_t dtor);
void tss_delete(tss_t key);
void *tss_get(tss_t key);
int tss_set(tss_t key, void *val);
```

Threads, séparation des états

initialisation

- faire le plus possible de l'initialisation dans le `main`
- si ce n'est pas possible `once_flag`

```
void call_once(once_flag *flag, void (*func)(void));
```

```
char* text = 0;
void myInitFunc(void) {
  text = malloc(bigNumber);
}

/* intérieur aux fonctions de thread */
static once_flag once = ONCE_FLAG_INIT;
call_once(&once, myInitFunc);
```

Threads, séparation des états

finalisation

```
char* text = 0;
void myFinalFunc(void) {
  free(text);
}
void myInitFunc(void) {
  atexit(myFinalFunc);
  text = malloc(bigNumber);
}

/* intérieur aux fonctions de thread */
static once_flag once = ONCE_FLAG_INIT;
call_once(&once, myInitFunc);
```