

Synchronization

François Trahay

Contents

Introduction	1
Atomic operations	2
Motivation	2
Can't we just use <code>volatile</code> ?	2
Atomic operations	4
Test and set	5
Compare And Swap (CAS)	6
Fetch and Add	9
Memory Fence (<i>Barrière mémoire</i>)	11
Synchronization primitives	11
Busy-waiting synchronization	11
Futex	13
Implementing a mutex using a futex	13
Implementing a monitor using a futex	16
Using synchronization	19
Deadlock	20
Lock granularity	20
Scalability of a parallel system	20
Bibliography	21

Introduction

- Objectives of this lecture:
 - How are synchronization primitives implemented?
 - How to do without locks?

If you want to study further synchronization primitives, and to understand memory models, the blog post *We Make a `std::shared_mutex` 10 Times Faster* <<https://www.codeproject.com/Articles/1183423/We-Make-a-std-shared->

mutex-10-Times-Faster> discusses in details atomic operations, instruction reordering, C++ memory model and various synchronization primitives.

Atomic operations

Motivation

- By default, an instruction modifying a variable is non-atomic
- example : `x++` gives :
 - `register = load(x)`
 - `register ++`
 - `x = store (register)`

→ Problem if the variable is modified by a other thread simultaneously

Can't we just use volatile ?

- Tells the compiler that the variable can change from one access to another:
 - modification by another thread
 - modification by a signal handler
- But `volatile` does not ensure atomicity

Here is an example of a program that may suffer from overly aggressive optimization by the compiler:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#ifdef USE_VOLATILE
volatile int a = 0;
#else
int a = 0;
#endif

void* thread1(void*arg) {
    while(a == 0) ;
    printf("Hello\n");
    return NULL;
}

void* thread2(void*arg) {
```

```

    a = 1;
    return NULL;
}

int main(int argc, char**argv) {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return EXIT_SUCCESS;
}

```

When compiled with the optimization level `-O0` (i.e. without any optimization), `thread1` spins waiting, and when `thread2` modifies the variable `a`, it unlocks `thread1` which displays Hello:

```

$ gcc -o volatile volatile.c -Wall -pthread -O0
$ ./volatile
Hello
$

```

When compiled with the optimization level `-O1`, the generated code no longer works:

```

$ gcc -o volatile volatile.c -Wall -pthread -O1
$ ./volatile
[waits indefinitely]
^C
$

```

Analyzing the code generated by the compiler reveals the problem:

```

$ gcc -o volatile volatile.c -Wall -pthread -O2
$ gdb ./volatile
[...]
(gdb) disassemble thread1
Dump of assembler code for function thread1:
0x0000000000000756 <+0>: auipc a5,0x2
0x000000000000075a <+4>: lw a5,-1778(a5) # 0x2064 <a>
0x000000000000075e <+8>: bnez a5,0x762 <thread1+12>
0x0000000000000760 <+10>: j 0x760 <thread1+10>
0x0000000000000762 <+12>: add sp,sp,-16
0x0000000000000764 <+14>: auipc a0,0x0
0x0000000000000768 <+18>: add a0,a0,36 # 0x788
0x000000000000076c <+22>: sd ra,8(sp)
0x000000000000076e <+24>: jal 0x620 <puts@plt>
0x0000000000000772 <+28>: ld ra,8(sp)

```

```

0x00000000000000774 <+30>:    li   a0,0
0x00000000000000776 <+32>:    add  sp,sp,16
0x00000000000000778 <+34>:    ret
nd of assembler dump.
$

```

We see here that at the address 0x760, the program jumps to the address 0x760. So it jumps in place indefinitely.

This is explained by the fact that the variable `a` is not `volatile`. The compiler therefore thinks it can optimize access to this variable: since the `thread1` function only accesses the variable in read-mode, the program loads the variable in a register (here, the `a5` register, see the instruction 0x75a), then consults the registry. When `thread2` modifies the variable `a`, the modification is therefore not perceived by `thread1`!

Declaring the variable as `volatile` forces the compiler to read the variable each time:

```

$ gcc -o volatile volatile.c -Wall -pthread -O2 -DUSE_VOLATILE=1
$ gdb volatile
(gdb) disassemble thread1
Dump of assembler code for function thread1:
0x00000000000000756 <+0>:    add  sp,sp,-16
0x00000000000000758 <+2>:    sd   ra,8(sp)
0x0000000000000075a <+4>:    auipc a4,0x2
0x0000000000000075e <+8>:    add  a4,a4,-1782 # 0x2064 <a>
0x00000000000000762 <+12>:   lw   a5,0(a4)
0x00000000000000764 <+14>:   beqz a5,0x762 <thread1+12>
0x00000000000000766 <+16>:   auipc a0,0x0
0x0000000000000076a <+20>:   add  a0,a0,34 # 0x788
0x0000000000000076e <+24>:   jal  0x620 <puts@plt>
0x00000000000000772 <+28>:   ld   ra,8(sp)
0x00000000000000774 <+30>:   li   a0,0
0x00000000000000776 <+32>:   add  sp,sp,16
0x00000000000000778 <+34>:   ret
End of assembler dump.

```

Here, the loop `while (a == 0)` is translated to the lines from 0x762 to 0x764. At each loop iteration, the value of `a` is loaded, then tested.

Atomic operations

- C11 provides a set of atomic operations, including
 - `atomic_flag_test_and_set`
 - `atomic_compare_exchange_strong`

- atomic_fetch_add
 - atomic_thread_fence
-

Test and set

```
_Bool atomic_flag_test_and_set(volatile atomic_flag* obj);
```

- sets a flag and returns its previous value

Performs atomically:

```
int atomic_flag_test_and_set(int* flag) {
    int old = *flag;
    *flag = 1;
    return old;
}
```

Implementing a lock:

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

Here is an example of a program using a `test_and_set` based lock:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER 1000000
#define NTHREADS 4

volatile int lock=0;

int x = 0;
#ifdef NOT_THREAD_SAFE

/* thread-unsafe version */
void do_lock() {
    while(lock) ;
    lock = 1;
}

void do_unlock() {
```

```

    lock = 0;
}

#else

/* thread-safe version */
void do_lock() {
    while(atomic_flag_test_and_set(&lock)) ;
}

void do_unlock() {
    lock = 0;
}

#endif      /* NOT_THREAD_SAFE */

void* thread_function(void* arg) {
    for(int i=0; i<NITER; i++) {
        do_lock();
        x++;
        do_unlock();
    }
    return NULL;
}

int main(int argc, char**argv) {
    pthread_t tids[NTHREADS];
    int ret;
    for(int i = 0; i<NTHREADS; i++) {
        ret = pthread_create(&tids[i], NULL, thread_function, NULL);
        assert(ret == 0);
    }
    for(int i = 0; i<NTHREADS; i++) {
        ret = pthread_join(tids[i], NULL);
        assert(ret == 0);
    }

    printf("x = %d\n", x);
    return EXIT_SUCCESS;
}

```

Compare And Swap (CAS)

`_Bool atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired);`

- compares *obj and *expected

- if equal, copy desired into *obj and return true
- else, copy the value of *obj into *expected and return false
- Performs atomically:

```
bool CAS(int* obj, int* expected, int desired) {
    if(*obj != *expected) {
        *expected = *obj;
        return false;
    } else {
        *obj = desired;
        return true;
    }
}
```

Here is an example of a program handling a *lock-free* list thanks to `compare_and_swap`:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER 1000000
#define NTHREADS 4

struct node {
    int value;
    struct node* next;
};

struct node *stack = NULL;

#ifdef NOT_THREAD_SAFE

/* thread-unsafe version */
void push(int value) {
    struct node* n = malloc(sizeof(struct node));
    n->value = value;
    n->next = stack;
    stack = n;
}

int pop() {
    struct node* n = stack;
```

```

int value = 0;
if(n) {
    value = n->value;
    stack = n->next;
    free(n);
}
return value;
}

#else

/* thread-safe version */
void push(int value) {
    struct node* n = malloc(sizeof(struct node));
    n->value = value;
    n->next = stack;

    int done = 0;
    do {
        done = atomic_compare_exchange_strong(&stack, &n->next, n);
    } while(!done);
}

int pop() {
    int value = 0;
    struct node* old_head = NULL;
    struct node* new_head = NULL;
    int done = 0;

    do {
        /* Warning: this function still suffers a race condition (search for
        * "ABA problem" for more information).
        * Fixing this would be too complicated for this simple example.
        */
        old_head = stack;
        if(old_head)
            new_head = old_head->next;
        done = atomic_compare_exchange_strong(&stack, &old_head, new_head);
    } while (!done);

    if(old_head) {
        value = old_head->value;
        free(old_head);
    }
    return value;
}

```



```

#endif          /* NOT_THREAD_SAFE */

_Atomic int sum = 0;
void* thread_function(void* arg) {
    for(int i=0; i<NITER; i++) {
        push(1);
    }

    int value;
    while((value=pop()) != 0) {
        sum+=value;
    }

    return NULL;
}

int main(int argc, char**argv) {
    pthread_t tids[NTHREADS];
    for(int i = 0; i<NTHREADS; i++) {
        pthread_create(&tids[i], NULL, thread_function, NULL);
    }
    for(int i = 0; i<NTHREADS; i++) {
        pthread_join(tids[i], NULL);
    }
    printf("sum = %d\n", sum);
    return EXIT_SUCCESS;
}

```

Fetch and Add

- C `atomic_fetch_add(volatile A* obj, M arg);`
 - replace `obj` with `arg+obj`
 - return the old value of `obj`
- Performs atomically:

```

int fetch_and_add(int* obj, int value) {
    int old = *obj;
    *obj = old+value;
    return old;
}

```

Here is an example of a program using `fetch_and_add` to atomically increment a variable:

```

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

#define NITER 1000000
#define NTHREADS 4

volatile int x = 0;

#ifdef NOT_THREAD_SAFE

/* thread-unsafe version */
void inc(volatile int * obj) {
    *obj = (*obj)+1;
}

#else

/* thread-safe version */
void inc(volatile int * obj) {
    atomic_fetch_add(obj, 1);
}

#endif /* NOT_THREAD_SAFE */

void* thread_function(void* arg) {
    for(int i=0; i<NITER; i++) {
        inc(&x);
    }
    return NULL;
}

int main(int argc, char**argv) {
    pthread_t tids[NTHREADS];
    for(int i = 0; i<NTHREADS; i++) {
        pthread_create(&tids[i], NULL, thread_function, NULL);
    }
    for(int i = 0; i<NTHREADS; i++) {
        pthread_join(tids[i], NULL);
    }

    printf("x = %d\n", x);
    return EXIT_SUCCESS;
}

```

Memory Fence (*Barrière mémoire*)

- C `atomic_thread_fence(memory_order order);`
 - performs a memory synchronization
 - ensures that all past memory operations are **visible** by all threads according to the memory model chosen (see C11 memory model)
-

Synchronization primitives

- Properties to consider when choosing a synchronization primitive
 - **Reactivity**: time spent between the release of a lock and the unblocking of a thread waiting for this lock
 - **Contention**: memory traffic generated by threads waiting for a lock
 - **Equity** and risk of *famine*: if several threads are waiting for a lock, do they all have the same probability of acquire it? Are some threads likely to wait indefinitely?
-

Busy-waiting synchronization

```
int pthread_spin_lock(pthread_spinlock_t *lock);
```

- tests the value of the lock until it becomes free, then acquires the lock

```
int pthread_spin_unlock(pthread_spinlock_t *lock);
```

- Benefits
 - Simple to implement (with `test_and_set`)
 - Reactivity
 - Disadvantages
 - Consumes CPU while waiting
 - Consumes memory bandwidth while waiting
-

It is also possible to implement a *spinlock* using an atomic operation:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>
```

```

#include <assert.h>

#define NITER 1000000
#define NTHREADS 4

struct lock {
    /* if flag=0, the lock is available
     * if flag=1, the lock is taken
     */
    volatile int flag;
};
typedef struct lock lock_t;

void lock(lock_t *l) {
    /* try to set flag to 1.
     * if the flag is already 1, loop and try again
     */
    while(atomic_flag_test_and_set(&l->flag)) ;
}

void unlock(lock_t *l) {
    l->flag = 0;
}

void lock_init(lock_t *l) {
    l->flag = 0;
}

lock_t l;
int x;

void* thread_function(void*arg){
    for(int i=0; i<NITER; i++) {
        lock(&l);
        x++;
        unlock(&l);
    }
    return NULL;
}

int main(int argc, char**argv) {
    lock_init(&l);

    pthread_t tids[NTHREADS];

```

```

int ret;
for(int i = 0; i<NTHREADS; i++) {
    ret = pthread_create(&tids[i], NULL, thread_function, NULL);
    assert(ret == 0);
}
for(int i = 0; i<NTHREADS; i++) {
    ret = pthread_join(tids[i], NULL);
    assert(ret == 0);
}

printf("x = %d\n", x);
printf("expected: %d\n", NTHREADS*NITER);
return EXIT_SUCCESS;
}

```

Futex

- *Fast Userspace Mutex*
 - System call allowing to build synchronization mechanisms in *userland*
 - Allows waiting without monopolizing the CPU
 - A futex is made up of:
 - * a value
 - * a waiting list
 - Available operations (among others)
 - * WAIT(int *addr, int value)
 - while(*addr == value) { sleep(); }
 - add the current thread to the waiting list
 - * WAKE(int *addr, int value, int num)
 - *addr = value
 - wake up num threads waiting on addr
-

Implementing a mutex using a futex

- mutex: an integer with two possible values: 1 (unlocked), or 0 (locked)
- mutex_lock(m):
 - *Test and unset* the mutex
 - if mutex is 0, call FUTEX_WAIT
- mutex_unlock(m):
 - Test and set the mutex
 - call FUTEX_WAKE to wake up a thread from the waiting list

Here is an example of a program implementing a mutex using futex:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <stdatomic.h>
#include <linux/futex.h>
#include <sys/time.h>
#include <sys/syscall.h>
#include <errno.h>
#include <assert.h>

#define NITER 1000000
#define NTHREADS 4

struct lock {
    int flag;
};
typedef struct lock lock_t;

static int futex(int *uaddr, int futex_op, int val,
                const struct timespec *timeout, int *uaddr2, int val3) {
    return syscall(SYS_futex, uaddr, futex_op, val,
                 timeout, uaddr2, val3);
}

void lock(lock_t *l) {
    while (1) {
        /* Is the futex available? */
        int expected = 1;
        if (atomic_compare_exchange_strong(&l->flag, &expected, 0))
            return; /* Yes */

        /* Futex is not available; wait */
        int s = futex(&l->flag, FUTEX_WAIT, 0, NULL, NULL, 0);
        if (s == -1 && errno != EAGAIN) {
            perror("futex_wait failed");
            abort();
        }
    }
}

void unlock(lock_t *l) {
    int expected = 0;
    atomic_compare_exchange_strong(&l->flag, &expected, 1);
    int s = futex(&l->flag, FUTEX_WAKE, 1, NULL, NULL, 0);
}

```

```

    if (s == -1) {
        perror("futex_wake failed");
        abort();
    }
}

void lock_init(lock_t *l) {
    l->flag = 1;
}

lock_t l;
int x;

void* thread_function(void*arg){
    for(int i=0; i<NITER; i++) {
        // printf("%d\n", i);
        lock(&l);
        x++;
        unlock(&l);
    }
    return NULL;
}

int main(int argc, char**argv) {
    lock_init(&l);

    pthread_t tids[NTHREADS];
    int ret;
    for(int i = 0; i<NTHREADS; i++) {
        ret = pthread_create(&tids[i], NULL, thread_function, NULL);
        assert(ret == 0);
    }
    for(int i = 0; i<NTHREADS; i++) {
        ret = pthread_join(tids[i], NULL);
        assert(ret == 0);
    }

    printf("x = %d\n", x);
    printf("expected: %d\n", NTHREADS*NITER);
    return EXIT_SUCCESS;
}

```

Implementing a monitor using a futex

- condition: a counter

```
struct cond {
    int cpt;
};

void cond_wait(cond_t *c, pthread_mutex_t *m) {
    int value = atomic_load(&c->value);
    pthread_mutex_unlock(m);
    futex(&c->value, FUTEX_WAIT, value);
    pthread_mutex_lock(m);
}

void cond_signal(cond_t *c) {
    atomic_fetch_add(&c->value, 1);
    futex(&c->value, FUTEX_WAKE, 0);
}
```

Here is an example of a program implementing a condition using futex:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/syscall.h>
#include <linux/futex.h>
#include <stdatomic.h>
#include <assert.h>

#define N 10

int n_loops = 20;

struct cond {
    int cpt;
};
typedef struct cond cond_t;

static int futex(int *uaddr, int futex_op, int val) {
    return syscall(SYS_futex, uaddr, futex_op, val, NULL, uaddr, 0);
}

void cond_init(cond_t *c) {
    c->cpt = 0;
}
```



```

void cond_wait(cond_t *c, pthread_mutex_t *m) {
    int cpt = atomic_load(&c->cpt);
    pthread_mutex_unlock(m);
    futex(&c->cpt, FUTEX_WAIT, cpt);
    pthread_mutex_lock(m);
}

void cond_signal(cond_t *c) {
    atomic_fetch_add(&c->cpt, 1);
    futex(&c->cpt, FUTEX_WAKE, 0);
}

struct monitor{
    int value;
    pthread_mutex_t mutex;
    cond_t cond;
};

int infos[N];
int i_depot, i_extrait;
int nb_produits = 0;
struct monitor places_dispo;
struct monitor info_prete;

void* function_prod(void*arg) {
    static _Atomic int nb_threads=0;
    int my_rank = nb_threads++;

    for(int i=0; i<n_loops; i++) {
        int cur_indice;
        int product_id;
        usleep(100);
        pthread_mutex_lock(&places_dispo.mutex);
        while(places_dispo.value == 0) {
            cond_wait(&places_dispo.cond, &places_dispo.mutex);
        }
        places_dispo.value--;
        cur_indice = i_depot++;
        i_depot = i_depot % N;

        product_id = nb_produits++;
        pthread_mutex_unlock(&places_dispo.mutex);
    }
}

```

```

        usleep(500000);
        printf("P%d produit %d dans %d\n", my_rank, product_id, cur_indice);

        pthread_mutex_lock(&info_prete.mutex);
        infos[cur_indice] = product_id;
        info_prete.value ++;
        cond_signal(&info_prete.cond);
        pthread_mutex_unlock(&info_prete.mutex);
    }
    return NULL;
}

void* function_cons(void*arg) {
    static _Atomic int nb_threads=0;
    int my_rank = nb_threads++;

    for(int i=0; i<n_loops; i++) {
        int cur_indice;
        int product_id;
        usleep(100);
        pthread_mutex_lock(&info_prete.mutex);
        while(info_prete.value == 0) {
            cond_wait(&info_prete.cond, &info_prete.mutex);
        }
        info_prete.value--;
        product_id = infos[i_extrait];
        cur_indice = i_extrait;
        i_extrait = (i_extrait+1) % N;
        pthread_mutex_unlock(&info_prete.mutex);

        usleep(100000);
        printf("C%d consomme %d depuis %d\n", my_rank, product_id, cur_indice);

        pthread_mutex_lock(&places_dispo.mutex);
        places_dispo.value ++;
        cond_signal(&places_dispo.cond);
        pthread_mutex_unlock(&places_dispo.mutex);
    }
    return NULL;
}

void init_monitor(struct monitor *m, int value) {
    m->value = value;
    pthread_mutex_init(&m->mutex, NULL);
}

```

```

    cond_init(&m->cond);
}

int main(int argc, char**argv) {
    init_monitor(&places_dispo, N);
    init_monitor(&info_prete, 0);
    i_depot = 0;
    i_extrait = 0;

    int nthreads_prod=2;
    int nthreads_cons=2;
    pthread_t tid_prod[nthreads_prod];
    pthread_t tid_cons[nthreads_cons];
    int ret;

    for(int i=0; i<nthreads_prod; i++) {
        ret = pthread_create(&tid_prod[i], NULL, function_prod, NULL);
        assert(ret == 0);
    }
    for(int i=0; i<nthreads_cons; i++) {
        ret = pthread_create(&tid_cons[i], NULL, function_cons, NULL);
        assert(ret == 0);
    }

    for(int i=0; i<nthreads_prod; i++) {
        ret = pthread_join(tid_prod[i], NULL);
        assert(ret == 0);
    }
    for(int i=0; i<nthreads_cons; i++) {
        ret = pthread_join(tid_cons[i], NULL);
        assert(ret == 0);
    }

    return EXIT_SUCCESS;
}

```

Using synchronization

- Classic problems:
 - *deadlocks*
 - lock granularity
 - scalability

Deadlock

- Situation such that at least two processes are each waiting for a non-shareable resource already allocated to the other
- Necessary and sufficient conditions (Coffman, 1971 (Coffman, Elphick, and Shoshani 1971))
 1. Resources accessed under mutual exclusion (non-shareable resources)
 2. Waiting processes (processes keep resources that are acquired)
 3. Non-preemption of resources
 4. Circular chain of blocked processes
- Strategies:
 - Prevention: acquisition of mutexes in the same order
 - Deadlock detection and resolution (eg. with `pthread_mutex_timedlock`)

Lock granularity

- Coarse grain locking
 - A lock protects a large portion of the program
 - Advantage: easy to implement
 - Disadvantage: reduces parallelism
- Fine grain locking
 - Each lock protects a small portion of the program
 - Advantage: possibility of using various resources in parallel
 - Disadvantages:
 - * Complex to implement without bug (eg. deadlocks, memory corruption)
 - * Overhead (locking comes at a cost)

Scalability of a parallel system

- Scalability = ability to reduce execution time when adding processing units
- Sequential parts of a program reduce the scalability of a program (Amdahl's law (Amdahl 1967))
- In a parallel program, waiting for a lock introduced sequentiality → Locks can interfere with scalability

The notion of scalability is discussed in more detail in the module CSC5001 *High Performance Systems*.

Bibliography

- Amdahl, Gene M. 1967. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities.” In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 483–85. ACM.
- Coffman, Edward G, Melanie Elphick, and Arie Shoshani. 1971. “System Deadlocks.” *ACM Computing Surveys (CSUR)* 3 (2): 67–78.