

Synchronization

François Trahay

Introduction

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

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

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) ;  
}
```

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;  
    }  
}
```

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;  
}
```


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

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

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);
}
```

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

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.