Lock implementations

Master in computer science of IP Paris

Master CHPS of Paris Saclay

Gaël Thomas

Atomic operation (informal definition)

- At high-level, an atomic operation is an operation that **seems** to execute instantaneously

- `pthread_mutex_lock` is a typical atomic operation that executes something like:

```
enum { FREE, BUSY };
typedef struct { int state; } pthread_mutex_t;
#define PTHREAD_MUTEX_INITIALIZER = { FREE };

void pthread_mutex_lock(pthread_mutex_t* mutex) {
  while(mutex->state == FREE) { }

  mutex->state = BUSY;
}
```

Another thread cannot take the lock here because `pthread_mutex_lock` executes atomically

How can we build atomic operation?

Problem: we cannot implement `pthread_mutex_lock` with `pthread_mutex_lock` since we are implementing `pthread_mutex_lock`!

Menu

How can we build atomic operation?

- Before everything, we need a model of machine
- At this step, we consider a simple machine model that:
  - Atomically reads and writes a machine word (32bits and 64bits)

```
                    int x = 0;

    Thread 1                Thread 2

  x = 0x101;              tmp = x;
```

Here: `tmp` = 0 or 0x101 (not 0x100 nor 0x001)

How can we build atomic operation?

- Before everything, we need a model of machine

- At this step, we consider a simple machine model that:
  - Atomically reads and writes a machine word (32bits and 64bits)

Th

x =

0x001)

Be careful!

In general, this hypothesis does not hold!

(it's true with a pentium, but not with all the possible existing or future processors)

How can we build atomic operation?

- Before everything, we need a model of machine
- At this step, we consider a simple abstract machine that:
  - Atomically reads and writes a machine word (32bits and 64bits)
  - Does not reorder the instructions

```
                  int x = 0;
                  int y = 0;

     Thread 1               Thread 2

   x = 0x101;            t2 = y;
   y = 0x202;            t1 = x;
```

Here, t2 = 0x202 => t1 = 0x101

How can we build atomic operation?

- Before everything, we need a model of machine

- At this step, we consider a simple abstract machine that:
  - Atomically reads and writes a machine word (32bits and 64bits)
  - Does not reorder the instructions

```
  Thr

x = 0x
y = 0x
```

Be careful!

In general, this hypothesis does not hold!

(for two writes, it's true with a pentium, but not for a read that succeeds a write!)

Menu

The bakery algorithm (Lamport 1974)

- Principle
  - Simulates a bakery where people are waiting to order
  - Each waiter has a number
  - A waiter can order if all the waiters with lowest number are already served

- Problem: how can we choose a number?
  - Idea: ask to the other waiters and choose the highest one + 1
  - New problem: if two waiters asks at the same time, they will have the same number
  - In this case, since we are polite, the oldest is served first (we suppose that birth dates are unique)

The bakery algorithm (Lamport 1974)

- We suppose our simple machine model

```
int entering[N]; /* initialized to false */
int num[N];      /* initialized to 0 */

void lock(int self) { /* thread number self calls lock */
  entering[self] = true;
  num[self] = 1 + max(num[0], …, num[N-1]);
  entering[self] = false;

restart:
  for(int i=0; i<N; i++) {
    if(entering[i]) { } /* wait until i receives its number */
    if(num[i] && num[i] < num[self]) goto restart;
    if(num[i] && num[i] == num[self] && i < self) goto restart;
  }
}

void unlock(int self) { num[self] = 0; } /* outside */
```

Why the entering variable

- Suppose that we don’t have entering
  - Processes 2 and 3 enter at the same time
  - Process 2 computes the max+1, but is preempted before writing it to num
  - Process 3 executes and computes the same max+1
  - Process 3 enters the critical section, but is preempted before unlock
  - Process 2 is elected and enters the critical section since 2 < 3
  - => we have two processes in the critical section

- Entering prevents the thread 3 to enters the critical section because thread 3 sees entering[2] = true

The bakery algorithm (Lamport 1974)

- Beautiful algorithm
  - Only requires write atomicity
    (and no reordering of the instructions)

- But the algorithm is slow
  - Reads at least 4 memory locations per thread
    - One to compute max
    - Two for entering
    - One to check that the thread has the maximum number
  - => (better) complexity in O(N) in term of reads where N is the number of threads

We need help from the hardware

- We need special instructions to optimize the lock implementation

- But before, we have to understand how a processor behaves
  a. The cache protocol
  b. Load and store atomicity
  c. Memory ordering

Menu

The cache protocol

- A cache of the processor contains copies of memory location
  - Cache lines of 64 bytes for most pentium
  - Implements a **read-write lock**
    - One writer or multiple readers
    - In case of read, the cache line is in the shared state (multiple readers)
    - In case of write, it is in the exclusive state (only one writer)

```
                    int tab[2];
  Thread 1                              Thread 2

tab[0] = 42;                        tab[1] = 666;
```

Cache line: [42 0] $\longleftrightarrow$ Cache line: [0, 666]

If multiple writers at the same time, almost impossible to not lose one of the writes

The cache protocol

- In case of store, the cache protocol acquires a cache line in exclusive state in order to ensure consistency

- Without exclusion during stores, the processor may lose stores

```
int tab[2];
```

Thread 1	**Thread 2**
`tab[0] = 42;`	`tab[1] = 666;`

Cache line: [42 0] $\longleftrightarrow$ Cache line: [0, 666]

If multiple writers at the same time, almost impossible to not lose one of the stores

Implementation of the cache protocol

- In case of load, if the line is not in the cache
  - Loads the line from another core or from the main memory
  - Ensures that other cores do not hold the line in exclusive state
  - Marks the cache line as shared
- In case of store, if the line is not in the cache
  - Loads the line from another core or from the main memory
  - Invalidates the other copies in the other cores
  - Marks the cache line as exclusive
- In case of store, if the line is in the cache but is shared
  - Invalidates the other copies in the other cores
  - Marks the cache line as exclusive

The cache protocol

- Consequence: if many threads running on different cores write the same cache line, the memory buses saturates

- Consequence for lock algorithms
  - Try to avoid many threads writing the same memory location

Menu

Load and store atomicity

- A load or a store of a machine word is not necessarily atomic
  - It's the case with a pentium
  - But not necessarily with any processor that may appear in the future!

- In order to ensure load and store atomicity in C
  - `atomic_load(&var)`: ensures load atomicity
  - `atomic_store(&var, value)` : ensures store atomicity
  - In this case, var should be declared as `_Atomic`, e.g., `int _Atomic var;`

- Note: these operations have also an effect on ordering

Menu

Memory ordering

- A processor may emit the instructions out-of-order
  (as soon as it ensures that a thread reads its own last write)

- A processor may reorder
  - Two stores on two different memory locations
    - `store @a1, v1`
    - `store @a2, v2`
  - Two loads on two different memory locations
    - `v1 = load @a1`
    - `v2 = load @a2`
  - A load after a store on two different memory locations
    - `store @a1, v1`
    - `v2 = load @a2`
  - A store after a load on two different memory locations

Each language and each processor has its own memory ordering model

- Pentium: total store order
  - Ensure atomicity for 64-bits loads and stores
  - Does not reorder a load after a load
  - Does not reorder a store after a store
  - Does not reorder a store after a load
  - But may reorder a load after a store

```
int hasMessage = false;
char* message = NULL;
```

```
        Thread 1                              Thread 2

message = "hello";               while(!hasMessage) { }
hasMessage = true;               printf(message);
```

Correct code (store after store in thread 1 and load after load in thread 2)

Each language and each processor has its own memory ordering model

- Java memory model
  - Any ordering is possible
    (except around volatile accesses and lock/unlock)

- ARM memory model
  - Weaker than TSO

```
int hasMessage = false;
char* message = NULL;
```

```
        Thread 1                           Thread 2

message = "hello";                while(!hasMessage) { }
hasMessage = true;                printf(message);
```

Possible segmentation fault in printf in Java or on a ARM

Preventing reordering with assembly instructions

- A processor provides special instructions

- For example, with a pentium
  - `mfence`: full memory fence prevents any reordering of loads or stores before or after the instruction

  - `lfence`: load fence prevents the reordering of the loads before or after the instruction (useful with special instructions that have a load semantic, e.g., `rdtsc`)
  - `sfence`: store fence prevents the reordering of the stores before or after the instruction (useful with special instructions that have a store semantic, e.g., `clwb`)

Preventing reordering in C

- The developer can explicitly specify the ordering semantic with atomic loads and stores
  - `value <- atomic_load_explicit(&addr,` **`order`**`)`
  - `atomic_store_explicit(&addr, value,` **`order`**`)`

- `order` can have the values
  - `memory_order_relaxed`
  - `memory_order_consume`
  - `memory_order_acquire`
  - `memory_order_release`
  - `memory_order_acq_rel`
  - `memory_order_seq_cst`
    (default value with `atomic_load` and `atomic_store`)

Relaxed semantic

- Any reordering is possible
  - Note that this is the case for the non-atomic operations in C

```
int _Atomic x = 0;
int _Atomic y = 0;

Thread 1
atomic_store_explicit(&x, 0x101, memory_order_relaxed); // A
atomic_store_explicit(&y, 0x101, memory_order_relaxed); // B

Thread 2
r2 = atomic_load_explicit(&y, memory_order_relaxed); // C
r1 = atomic_load_explicit(&x, memory_order_relaxed); // D
```

- Possible values: r1 = 0 et r2 = 0x101
  - B executed before A => B C D A scheduling
  - D executed before C => D A B C scheduling

Relaxed

Important

Here, r0 and r1 can only have the values 0 or 0x101

If we don't use atomic operations, r0 and r1 can additionally have the values 0x100 or 0x001

- A … n C

```
int
int

Threa
atomic_store_explicit(&x, 0x101, memory_order_relaxed); // A
atomic_store_explicit(&y, 0x101, memory_order_relaxed); // B
```

```
Thread 2
r2 = atomic_load_explicit(&y, memory_order_relaxed); // C
r1 = atomic_load_explicit(&x, memory_order_relaxed); // D
```

- Possible values: r1 = 0 et r2 = 0x101
  - B executed before A => B C D A scheduling
  - D executed before C => D A B C scheduling

Release-acquire semantic

- Principle
  - Each visible effect that precedes a store in release is visible before the corresponding load
  - Each visible effect that succeeds a load in acquire is NOT visible before the correspond store in release

```
int _Atomic x = 0; int y = 0; int z = 0;

Thread 1
z = 1; // A
y = 17; // B
atomic_store_explicit(x, 42, memory_order_release); // C

Thread 2
r1 = atomic_load_explicit(x, memory_order_acquire); // D
r2 = y; // E
r3 = z; // F
```

If r1 == 42, then y==17 and z==1

Release-consume semantic

- As release-acquire, but only for variables that “carry” a dependency
  - So confusing that wrongly implemented in many compilers
  - The use of release-consume is currently discouraged!

```
int _Atomic x = 0; int y = 0; int z = 0;

Thread1:
x = 42; // A
y = 42; // B
atomic_store_explicit(&z, x, memory_order_release); // C

Thread2:
r1 = atomic_load_explicit(&z, memory_order_consume); // D
r2 = y; // E
r3 = x; // F
```

If r1 == 42, then r3 = 42 (because of the dependency between z and x in C), but r2 may be equal to 0

The sequential consistency semantic

- As release-acquire, but also ensures that all the threads see the atomic stores in sequential consistency in the same order

```
int _Atomic x = 0; int _Atomic y = 0;

Thread 1: atomic_store(&x, 1); // A
Thread 2: atomic_store(&y, 1); // B
Thread 3: r1 = atomic_load(&x); r2 = atomic_load(&y);
Thread 4: r3 = atomic_load(&x); r4 = atomic_load(&y);
```

If r1 = 1 and r2 = 0, then

- A executed before B for thread 3
- Because of sequential consistency, it’s also the case for thread 4
- r3 = 0 and r4 = 1 is thus impossible (B before A for thread 4)

With only release-acquire, we could have r3 = 0 and r4 = 1

Note

- In the remainder of the lectures, in order to simplify the codes
  - We don't explicitly specify atomic loads and stores
    `x = 0x101` means `atomic_store(&x, 0x101)`
  - We always suppose the sequential consistency semantic (even when a weaker semantic leads to a correct behavior)

- In the labs, you will
  - Have to explicitly use `atomic_load`/`atomic_store` for the shared variables
  - Have to try to identify if a weaker semantic such as acquire-release or even relaxed leads to a correct behavior

And now...

- We are now able to implement the bakery algorithm in C 😀

- In order to implement efficient lock algorithms, we still need load-modify-store operations that atomically
  - Load a value
  - Modify the value
  - Store the value

- Three important operations
  - `atomic_exchange`
  - `atomic_fetch_add`
  - `atomic_compare_exchange_strong`
  - Note: add `_explicit` to specify the memory order semantic

Menu

The tool: atomic_exchange

- Atomically exchange a value

```
type atomic_exchange(type _Atomic* addr, type value) {
  type res = *addr;
  *addr = value;
  return res;
}
```

$\}$ Executed atomically

Implementation of an atomic load-modify-store operation

- On a pentium, relies on the cache protocol
  - Acquire the cache line in exclusive mode
  - And “locks” the cache line in the cache during the execution of the load-modify-store instruction
  - Another core has thus to wait to acquire the cache line in shared or exclusive state

Implementation of an atomic loa

■ C

on of

shared

Important

The atomicity of load-modify-store operation does not require a global consensus with the other cores. It only consists in locking the cache line in the local L1 cache of the core during few cycles.

As a consequence, an atomic operation does not have a performance cost because of the atomicity!

The cost comes from: (i) the write that invalidates the other copies and (ii) the memory ordering that prevents out-of-order execution

The spinlock

- The spinlock is the most simple lock implementation
  - Principle: spins while a lock is in the BUSY state
  - Only requires the `atomic_exchange` operation

```
enum { FREE, BUSY };
int _Atomic lock = FREE;

void lock(int _Atomic* lock) {
  while(atomic_exchange(lock, BUSY) != FREE) { }
}

void unlock(int _Atomic* lock) {
  atomic_store(lock, FREE);
}
```

Note: the acquire-release semantic gives a correct behavior

The spinlock

- The spinlock is the most simple lock implementation
  - Principle: spins while a lock is in the BUSY state
  - Only requires the `atomic_exchange` operation
- Very efficient if the lock is almost always FREE
- Very inefficient in case of contention
  - The cache line that holds the lock variable continuously bounces between the cores
- Recall the cache protocol: in case of write, a core acquires a cache line in the exclusive state and invalidates thus the copies in the other cores

Menu

The tool: atomic_fetch_add

- Atomically adds a value to a memory location and returns the original value

```
type atomic_fetch_add(type _Atomic* addr, type n) {
  type res = *addr;
  *addr += n;
  return res;
}
```

Executed atomically

The ticket lock

- Very efficient lock implementation used in the Linux kernel
  - Based on `atomic_fetch_add`
  - Simulates a ticket with a number such as the one used at a post office

  - A client takes a ticket with a number, which atomically increments a counter for the next client
  - The postman increments another counter on a screen when a client leaves the post office
  - When the counter given by the ticket is equal to the counter given by the screen, the client is served

- Conceptually close to the Bakery algorithm

The ticket lock

```
struct ticket_lock {
  int _Atomic ticket;
  int _Atomic screen;
}; /* initialized to (0, 0) */

void lock(struct ticket_lock* t) {
  int my = atomic_fetch_add(&t->ticket, 1);
  while(atomic_load(&t->screen) < my) { }
}

void unlock(struct ticket_lock* t) {
  atomic_fetch_add(&t->screen, 1);
}
```

The ticket lock

The threads spin with a load operation and not with a load-modify-store operation, which avoids the cache line bounces caused by the cache line invalidations

```
struct ticket_lock {
  int _Atomic ticket;
  int _Atomic screen;
}; /* initialized to (0, 0) */

void lock(struct ticket_lock* t) {
  int my = atomic_fetch_add(&t->ticket, 1);
  while(atomic_load(&t->screen) < my) { }
}

void unlock(struct ticket_lock* t) {
  atomic_fetch_add(&t->screen, 1);
}
```

The ticket lock

```
struct ticket_lock {
  int _Atomic ticket;
  int _Atomic screen;
}; /* initialized to (0, 0) */

void lock(struct ticket_lock* t) {
  int my = atomic_fetch_add(&t->ticket, 1);
  while(atomic_load(&t->screen) < my) { }
}

void unlock
  atomic_fe
}
```

The threads spin with a load operation and not with a load-modify-store operation, which avoids the cache line bounces caused by the cache line invalidations

However, all the threads spin while loading the same memory location

We can probably do better!

Menu

The tools: atomic_compare_exchange_strong

- Like exchange, but only if the variable has a given value

```
bool atomic_compare_exchange_string(type _Atomic* addr,
                                     type* expected,
                                     type value) {
  if(*addr == *expected) { /* success */
    *addr = value;         /* exchange */
    return true;
  } else {                 /* fail */
    *expected = *addr;/* replace *expected by actual */
    return false;
  }
}
```

The tools: the _Thread_local storage

- We often need global per-thread variable
  - To store a thread number
  - For thread-specific data structures

- The `_Thread_local` storage class specifier
  - Define a global variable
  - With a per-thread semantic (one variable per thread)

- Example
  - `_Thread_local int myId;`

The MCS lock [ASPLOS'91]

- Principle:
  - Create a FIFO of processes that waits for the lock
  - The lock owner wakes up the next in the list
- Advantages
  - Totally fair
  - Each waiter spins alone on its memory location
  - The thread owner only wakes up the next in the FIFO queue
- Drawback
  - Subject to the convoy effect, especially when the process contains more threads than the number of cores

John M. Mellor-Crummey, Michael L. Scott: Synchronization without Contention. ASPLOS 1991

The MCS lock [ASPLOS'91]

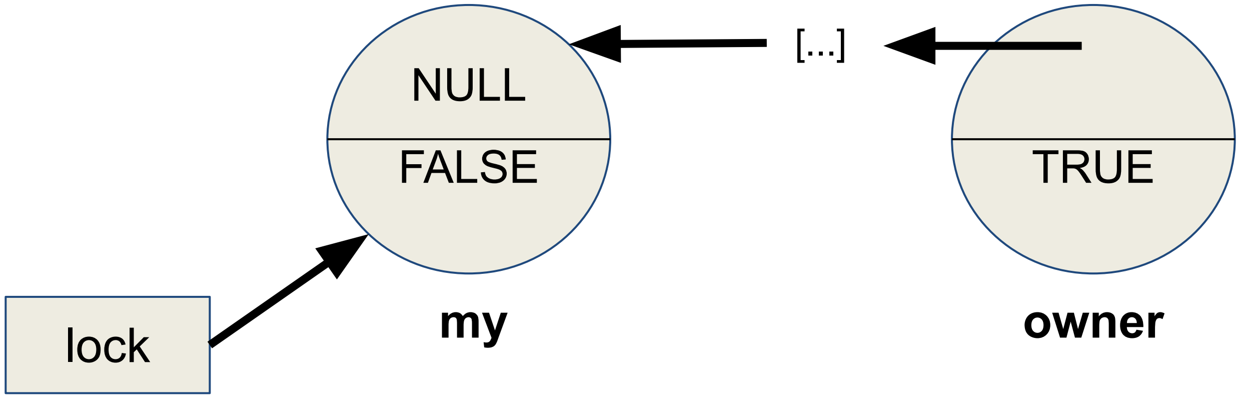
```
struct node { struct node* _Atomic next; bool _Atomic isFree; };
_Thread_local struct node my;
struct node* _Atomic lock = NULL;

void my_lock() {
  my.next = NULL;    my.isFree = false;
  struct node* p = atomic_exchange(&lock, &my);
  if(p) {
    atomic_store(&p->next, &my);
    while(!atomic_load(&my.isFree)) { }
  }
}
```

NULL means that the lock is free

If the lock is already taken

add `my` in the waiting queue



The MCS lock [ASPLOS'91]

```
void my_lock() {
  my.next = NULL;    my.isFree = false;
  struct node* p = atomic_exchange(&lock, &my);
  if(p) {
    atomic_store(&p->next, &my);
    while(!atomic_load(&my.isFree)) { }
  }
}

void my_unlock() {
  struct node* expected = &my;
  if(!atomic_load(&my.next)
      && atomic_compare_exchange_strong(&lock, &expected, NULL))
    return;
  while(!atomic_load(&my.next)) { }
  atomic_store(&my.next->isFree, true);
}
```

If `my.next` is still null

And if `my` is still at the head of the queue

No waiter => return

Wait while the next waiter has not yet installed the `next` pointer

Release the next waiter

To take away

- The Bakery algorithm
- `atomic_load` and `atomic_store`
- Memory ordering
  - Relaxed
  - Release-acquire
  - Release-consume
  - Sequential consistency
- atomic load-modify-store operations
  - Exchange to implement the spinlock
  - Fetch and add to implement the ticket lock
  - Compare and swap to implement the MCS lock