Multi-core CPUs are today’s norm, many-core CPUs will come tomorrow.

To take advantage of such a hardware, parallel (multi-threaded) programs must be run on them.

It is not useful when the threads are completely independent, i.e. threads have to communicate (synchronize).

Basic forms of synchronization:
- Mutual exclusion (e.g. access to shared data)
- Producer-consumer (e.g. database waits for requests)
- ...
Naive synchronization
Mutual exclusion

Data should be modified at most by one thread at a time:

```c
bool locked;

void func() {
    while (locked == true) /* busy wait */;
    locked = true;
    data++;
    locked = false;
}
```

**Terminology:** code in the “locked” region is called *critical section*

**Problems:**

1. Checking and setting the lock is not atomic
2. Compiler can optimize out all accesses to `locked`
3. Compiler can move access to data out of critical section
4. Hardware can reorder memory accesses even if compiler does not
5. Can easily deadlock
6. Busy waiting wastes energy
Atomic operations

- Example of non-atomic increment:
  - C expression: `data++`;
  - Assembler (x86): `inc ($data)` – uninterruptible
  - Hardware: memory bus read, ALU, memory bus write

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>bus read</td>
<td>bus read</td>
<td>0</td>
</tr>
<tr>
<td>ALU</td>
<td>ALU</td>
<td>1</td>
</tr>
<tr>
<td>bus write</td>
<td>bus write</td>
<td>1</td>
</tr>
</tbody>
</table>

- Atomic operations ensure that the operation (typically read-modify-write) is atomic (uninterruptible) even at the hardware (bus) level.
  - compare-and-swap/CAS instruction (x86: `cmpxchg`)

```c
void lock() {
    while (locked == true) /* busy wait */;
    locked = true;
}
```

```c
void lock() {
    while (__atomic_exchange_n(&locked, true, ...)) == true) /* busy wait */;
}
```
Atomic operations in C and C++

- For long time, atomic operations were not standardized in C/C++
  - Solution: Incompatible compiler extensions, inline assembler
- C11, C++11 introduced thread-aware memory model and defined platform independent atomic operations
- C11: stdatomic.h, atomic_* functions
- C++11
  - std::atomic template
  - std::atomic<int> x;
    x++; // atomic increment
Compiler optimizations

```c
bool locked;

while (locked) {}
locked = true;
data++;
locked = false;
```

```c
#define barrier() \
asm volatile("" : : : "memory"

volatile bool locked;

while (locked) {}
locked = true;
barrier();
data++;
barrier();
locked = false;
```

- Compiler expects the memory is only modified by the program being compiled.
- Locked seems to be useless ⇒ optimize out.
- Compiler is free to reorder operations as long as the result of the computation is the same.
Defining the variable volatile makes all accesses “volatile” i.e. slow.

Sometimes, we need only certain accesses to have volatile semantics and the rest can be optimized:

```c
#define ACCESS_ONCE(x) (*(volatile typeof(x) *)&(x))
#define LOAD_SHARED(p) ACCESS_ONCE(p)
#define STORE_SHARED(x, v) ({ ACCESS_ONCE(x) = (v); })
#define barrier() asm volatile("" : : : "memory")
```

The macro `barrier` is only a compiler barrier, not hardware barrier, i.e., the compiler will not reorder generated instructions.
Different CPU architectures implement different memory consistency models

Some operations can be reordered with respect to other operations

<table>
<thead>
<tr>
<th>Type</th>
<th>Alpha</th>
<th>ARMv7</th>
<th>PA-RISC</th>
<th>POWER</th>
<th>SPARC RMO</th>
<th>SPARC PSO</th>
<th>SPARC TSO</th>
<th>x86</th>
<th>x86 ostore</th>
<th>AMD64</th>
<th>IA-64</th>
<th>z/Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Loads stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Stores stores</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Stores loads</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Atomic loads</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Atomic stores</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>Dependent loads</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Incoherent inst. cache pipeline</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
</tr>
</tbody>
</table>

x86 can reorder stores after loads, i.e. data can be read before other CPUs see locked set to true!

Why? Stores may have to wait for cache-line ownership. Not waiting with subsequent reads improves performance.

Solution: Insert memory barrier instructions.
  - e.g. mfence, lfence on x86
Specifying memory ordering requirements in C/C++

```cpp
std::atomic<int> x;
x.load(order);
w.store(0, order);
```

- `order` specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation
  - relaxed: no overhead, no order guarantee
  - consume
  - acquire
  - release
  - acq_rel,
  - seq_cst: high overhead, sequential consistency

- Depending on the CPU architecture, different orders cause the compiler to generate barrier instructions (e.g., lfence on x86)
Cost of atomic operations & barriers

16-CPU 2.8GHz Intel X5550 (Nehalem) System

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost (ns)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock period</td>
<td>0.4</td>
<td>1.0</td>
</tr>
<tr>
<td>“Best-case” CAS</td>
<td>12.2</td>
<td>33.8</td>
</tr>
<tr>
<td>Best-case lock</td>
<td>25.6</td>
<td>71.2</td>
</tr>
<tr>
<td>Single cache miss</td>
<td>12.9</td>
<td>35.8</td>
</tr>
<tr>
<td>CAS cache miss</td>
<td>7.0</td>
<td>19.4</td>
</tr>
<tr>
<td>Single cache miss (off-core)</td>
<td>31.2</td>
<td>86.6</td>
</tr>
<tr>
<td>CAS cache miss (off-core)</td>
<td>31.2</td>
<td>86.5</td>
</tr>
<tr>
<td>Single cache miss (off-socket)</td>
<td>92.4</td>
<td>256.7</td>
</tr>
<tr>
<td>CAS cache miss (off-socket)</td>
<td>95.9</td>
<td>266.4</td>
</tr>
</tbody>
</table>

Source: Paul E. McKenney, IBM

- Atomic operations are costly (here 19–266 times slower than non-atomic operations)
- Barriers are typically cheaper (weak barriers more that full barriers)
Cost of atomic operations & laws of physics

Speed of light RT in 1 cycle @ 3 GHz = 5 cm
Speed of electrons in transisors: 0.03–0.3C

Every CPU experiences a cache miss, because other CPUs access the variable as well

No cache miss ⇒ much faster
Naive synchronization » Problems

Locking overhead

```c
pthread_mutex_lock(mutex);
x++;
pthread_mutex_unlock(mutex);
```

- Uncontended case: during lock(), mutex is not in the cache, during unlock() it is
- Contended case: mutex is not in the cache even during unlock, because there is (probably) another CPU trying to lock the mutex and thus "stealing" the lock from mutex-owner’s cache

**Single-instruction critical sections protected by multiple locks**

![Diagram showing uncontended and contended cases with locking overhead.](attachment:image.png)
Deadlock

- Example:
  - Single-core system
  - Two threads low- and high-priority

  LP_thread       HP_thread
  ~~~~~~~~~~~~   ~~~~~~~~~~~
  lock();
  data++;
  → preemption →
  deadlock();

- **Solution:** When the lock is not available, ask the OS scheduler to put your thread to sleep and wake you up after the lock is available
  - Problem: atomicity of checking the lock and going to sleep
  - Requires implementation in the OS kernel
Each system call adds overhead (≈ 100 cycles on modern HW)

It is preferable to use “fine-grain” locking, i.e. locks protect as little data as possible to prevent lock contention.

If fine-grain locking is effective the lock is not contended and threads rarely have to sleep, but always pay the syscall overhead!

That’s not efficient – the solution in Linux is called futex.
Futex

Fast Userspace Mutex

- Uncontended mutex never goes to kernel
- It uses atomic instruction `cmpxchg(val, expct, new) → prev`
- `futex_wait()` and `futex_wake()` are system calls

```cpp
class mutex {
public:
    mutex () : val (0) { }

    void lock () {
        int c;
        if ((c = cmpxchg (val, 0, 1)) != 0) {
            if (c != 2)
                c = xchg (val, 2);
            while (c != 0) {
                futex_wait (&val, 2);
                c = xchg (val, 2);
            }
        }
    }

    void unlock () {
        if (atomic_dec (val) != 1) {
            val = 0;
            futex_wake (&val, 1);
        }
    }

private:
    int val;
};
```

Futex uses

Futex primitive can be used to implement the following higher-level synchronization mechanisms:

- Mutexes
- Semaphores
- Conditional variables
- Thread barriers
- Read-write locks
The problem of mutex

Mutual exclusion in massively parallel **read-mostly workload**

1. Lock/unlock overhead
2. Dead time during updates

![Diagram showing CPU usage](image-url)
Read-Write lock

- Update blocks readers
- Can be implemented on top of mutex(es)
We want this

- **Updater does not block readers**
- **Is that possible?**
Read-copy-update (RCU) is scalable – typically up to hundreds or thousands of CPUs.

- Read-side scalability of various synchronization primitives
- RCU is **scalable** – typically up to hundreds or thousands of CPUs
- Locking does not scale
Read-Copy-Update (RCU)

Updating RCU-based list

1. Original list
2. Copy B
3. Update B to D
4. Make the updated element visible to readers
5. Wait after all readers stop accessing B and free it
Main mechanisms of RCU

1. Publishing of updates (34)
   - Ensure that updated data reach memory before the updated pointer
   - Compiler and memory barrier

2. Accessing new versions of data (how readers traverse the list)
   - Ensure that we see all the updates made before publishing
   - Compiler and memory barrier

3. Waiting for all readers to finish
   - The tricky part!
   - No explicit (and expensive) tracking of each reader (e.g. no reference counting)
   - RCU uses indirect way of determining the end of all read-side sections
   - In certain implementations (QSBR) read-side has zero overhead
   - Note: In Java, this is not needed, because objects are freed by the garbage collector. However, garbage collection has its overhead. RCU allows to have zero overhead.
**RCU concepts and API**

- **Read-side critical section**
  - `rcu_read_lock()`/`unlock()`

- **Quiescent state**
  - code outside r.s.c.s.

- **rcu_dereference()**

- **rcu_assign_pointer()**

- **Grace period**
  - started and waited by `synchronize_rcu()`

- **Must not happen!**
rcu_read_lock(); /* Start critical section. */
p = rcu_dereference(cptr);
/* *p guaranteed to exist. */
do_something_with(p);
rcu_read_unlock(); /* End critical section. */
/* *p might be freed!!! */

- rcu_read_lock()/unlock() and rcu_dereference() are cheap, sometimes \textit{nop}.
- Updaters are more heavy-weight.
RCU updater

```c
pthread_mutex_lock(&updater_lock); // not needed if there is
    // just one updater */
old_p = cptr;
    // copy if needed */
rcu_assign_pointer(cptr, new_p); // update */
pthread_mutex_unlock(&updater_lock);
synchronize_rcu(); // Wait for grace period
free(old_p);
```
How does it work?

- Many implementations possible
- Trade-off between read-side overhead and constraints of application structure
Quiescent-state based reclamation (QSBR)

// Protects registry from concurrent accesses
pthread_mutex_t rcu_gp_lock = PTHREAD_MUTEX_INITIALIZER;

LIST_HEAD(registry);

struct rcu_reader {
    unsigned long ctr;
    char need_mb;
    struct list_head node;
    pthread_t tid;
};

/* per-thread variable */
struct rcu_reader __thread rcu_reader;

void rcu_register_thread(void) {
    rcu_reader.tid = pthread_self();
    mutex_lock(&rcu_gp_lock);
    list_add(&rcu_reader.node, &registry);
    mutex_unlock(&rcu_gp_lock);
    rcu_thread_online();
}

void rcu_unregister_thread(void) {
    rcu_thread_offline();
    mutex_lock(&rcu_gp_lock);
    list_del(&rcu_reader.node);
    mutex_unlock(&rcu_gp_lock);
}

#define RCU_GP_ONLINE 0x1
#define RCU_GP_CTR 0x2

// global counter
unsigned long rcu_gp_ctr = RCU_GP_ONLINE;

static inline void rcu_read_lock(void) {}
static inline void rcu_read_unlock(void) {}

/* Every thread must call this function periodically */
/* * outside of read-side critical section. */
static inline void rcu_quiescent_state(void) {
    smp_mb();
    STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
}

/* call before blocking system call */
static inline void rcu_thread_offline(void) {
    smp_mb();
    STORE_SHARED(rcu_reader.ctr, 0);
}

/* call after return from blocking system call */
static inline void rcu_thread_online(void) {
    STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
}
void synchronize_rcu(void) {
    unsigned long was_online;
    was_online = rcu_reader.ctr;
    smp_mb();
    if (was_online)
        STORE_SHARED(rcu_reader.ctr, 0);
    mutex_lock(&rcu_gp_lock);
    update_counter_and_wait();
    mutex_unlock(&rcu_gp_lock);
    if (was_online)
        STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
    smp_mb();
}

static void update_counter_and_wait(void) {
    struct rcu_reader *index;
    STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr + RCU_GP_CTR);
    barrier();
    list_for_each_entry(index, &registry, node) {
        while (rcu_gp_ongoing(&index->ctr))
            msleep(10);
    }
}

static inline int rcu_gp_ongoing(unsigned long *ctr) {
    unsigned long v;
    v = LOAD_SHARED(*ctr);
    return v && (v != rcu_gp_ctr);
}

Properties:

- Grace periods are not shared
- Long waiting ⇒ higher memory consumption
- Works only on 64-bit architectures – the counter must not overflow
Genereal-purpose RCU

Properties:
- Does not restrict application structure
- Higher read-side overhead (still less than typical locks)

```c
#define RCU_GP_CTR_PHASE 0x10000
#define RCU_NEST_MASK 0x0ffff
#define RCU_NEST_COUNT 0x1

unsigned long rcu_gp_ctr = RCU_NEST_COUNT;

static inline void rcu_read_lock(void) {
    unsigned long tmp;
    tmp = rcu_reader.ctr;
    if (!(tmp & RCU_NEST_MASK)) {
        STORE_SHARED(rcu_reader.ctr, LOAD_SHARED(rcu_gp_ctr));
        smp_mb();
    } else {
        STORE_SHARED(rcu_reader.ctr, tmp + RCU_NEST_COUNT);
    }
}

static inline void rcu_read_unlock(void) {
    smp_mb();
    STORE_SHARED(rcu_reader.ctr, rcu_reader.ctr - RCU_NEST_COUNT);
}
```
void synchronize_rcu(void)
{
    smp_mb();
    mutex_lock(&rcu_gp_lock);
    update_counter_and_wait();
    barrier();
    update_counter_and_wait();
    mutex_unlock(&rcu_gp_lock);
    smp_mb();
}

static void update_counter_and_wait(void)
{
    struct rcu_reader *index;
    STORE_SHARED(rcu_gp_ctr, rcu_gp_ctr ^ RCU_GP_CTR_PHASE);
    barrier();
    list_for_each_entry(index, &registry, node) {
        while (rcu_gp_ongoing(&index->ctr))
            msleep(10);
    }
}

static inline int rcu_gp_ongoing(unsigned long *ctr)
{
    unsigned long v;
    v = LOAD_SHARED(*ctr);
    return (v & RCU_NEST_MASK) && ((v ^ rcu_gp_ctr) & RCU_GP_CTR_PHASE);
}
Update benchmarks

Fig. 9. Update Overhead, 8-core Intel Xeon, Logarithmic Scale

Fig. 10. Update Overhead, 64-core POWER5+, Logarithmic Scale

Fig. 11. Impact of Update-Side Critical Section Length on Read-Side, 8-core Intel Xeon, Logarithmic Scale

Fig. 12. Impact of Update-Side Critical Section Length on Read-Side, 64-core POWER5+, Logarithmic Scale
Conclusion

- RCU is scalable synchronization mechanism for hundreds/thousands of CPUs and read-mostly workload
- We have seen a RCU-based implementation of single-linked list, but many other common data structures can be implemented in RCU-compatible way

References

  https://www.efficios.com/publications
- Paul E. McKenney, What Is RCU? Guest Lecture for Technische Universität Dresden
  http://www2.rdrop.com/users/paulmck/RCU/RCU.2014.05.18a.TU-Dresden.pdf