Contents of Lecture 7

- C11 Threads 2(2)
These and the following atomic operations are called functions in the standard but are macros.

- *A* refers to an atomic type, e.g. `_Atomic int`.
- *C* refers to the corresponding non-atomic type, i.e. `int`.
- The atomic exchange function writes a new value and returns the old value pointed to by *ptr*.

- `C atomic_exchange_explicit(volatile A* ptr, C value, memory_order order);`
- `C atomic_exchange(volatile A* ptr, C value);`
Atomic Compare and Exchange

- There are two versions: strong and weak.
- The weak may fail (and let you know) and must therefore be used in a loop.
- The functions compare the value at the location pointed to by ptr with an expected value and if they are equal writes a new value.
- The result of the comparison is returned.
- If the values are not equal, the current value is copied to expected, or actually to where the pointer expected points.
- The strong functions behave as:

  ```c
  if (*ptr == *expected)
      *ptr = value;
  else
      *expected = *ptr;
  ```
Atomic Compare and Exchange Function Prototypes

```c
bool atomic_compare_exchange_strong_explicit(
    volatile A* ptr,
    C* expected,
    C value,
    memory_order success,
    memory_order failure);
```

```c
bool atomic_compare_exchange_weak_explicit(
    volatile A* ptr,
    C* expected,
    C value,
    memory_order success,
    memory_order failure);
```

```c
bool atomic_compare_exchange_strong(
    volatile A* ptr,
    C* expected,
    C value);
```

```c
bool atomic_compare_exchange_weak(
    volatile A* ptr,
    C* expected,
    C value);
```

For the explicit functions, memory is affected according to parameters success and failure, depending on the result of the comparison.
The weak compare and exchange functions may fail spuriously, which means they fail to perform the compare and exchange and ”give up”. If so, the return value is guaranteed to be false, and the current value is not copied to what ptr points.

The weak forms allow faster implementation on machines with load-locked/load-linked/load-and-reserve and store conditional instructions — instead of atomic compare and exchange.

The load-locked type of instructions are described below but the idea is to split the atomic operation into two instructions.

A processor $P$ first performs a load-locked and then a store conditional.

If a different processor $Q$ performs a store between the load-locked and store conditional made by $P$, then the store conditional made by $P$ fails.
A spin lock has no waiting queue with sleeping threads

Assume 0 means free and 1 means locked

```c
_Atomic int a;
int expected;
int value;

value = 1; // want to lock it
do
    expected = 0; // hope for unlocked
while (!atomic_compare_exchange_weak(&a, &expected, value));
```
Atomic Fetch and Modify Functions

- These functions atomically read-compute-modify an atomic object.

<table>
<thead>
<tr>
<th>computation</th>
<th>C operator</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>atomic_fetch_add_explicit</td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td>atomic_fetch_sub_explicit</td>
</tr>
<tr>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor</td>
<td>^</td>
<td>atomic_fetch_xor_explicit</td>
</tr>
<tr>
<td>and</td>
<td>&amp;</td>
<td>atomic_fetch_and_explicit</td>
</tr>
</tbody>
</table>

- Plus the usual non-explicit functions, for example:
  C atomic_fetch_add(volatile A* ptr, M value); adds value to what A points to.
  
- If A is arithmetic then M is C and if it’s atomic_address then M is ptr_diff_t.

- The value of *ptr before the operation is returned.
The type atomic_flag is the atomic type for the classic test-and-set operation.

```c
bool atomic_flag_test_and_set(
    volatile atomic_flag* ptr);
```

```c
bool atomic_flag_test_and_set_explicit(
    volatile atomic_flag* ptr,
    memory_order order);
```

These functions set the flag if it was cleared.

If it already was set, it may be written again but that would be a poor implementation due to cache effects if multiple processors are waiting for the flag to be cleared.

The old value is returned.
Atomic Flag

- The atomic flag type and functions are the minimal hardware supported atomic operations.
- All others can be implemented using these.
- However, better performance can be achieved with more hardware support.
bool atomic_flag_clear(
    volatile atomic_flag* ptr);

bool atomic_flag_clear_explicit(
    volatile atomic_flag* ptr,
    memory_order order);

- These functions clear the flag.
- The order may neither be memory_order_acquire nor memory_order_acq_rel
Initially some in the standardization committee wanted to standardize on sequential consistency.

Fortunately, C11 has standardized on a relaxed memory model for non-atomic objects.

This is partly based on input from Linux kernel developers.

Thus, to make an assignment visible to another thread requires synchronization between the writing and reading threads.

There are three main kinds of synchronization and we will start with mutex unlock/lock.
Mutex Unlock/Lock

- A mutex in C11 is called `mtx_t`
  ```c
  int    a;
  
  Thread 1            Thread 2
  mtx_lock(&m);
  a = 1;
  mtx_unlock(&m);

  mtx_lock(&m);
  printf("a = %d\n", a);
  mtx_unlock(&m);
  ```

- The unlock by Thread 1 and lock by Thread 2 make the write of `a` visible to Thread 2.
- A part of the mutex unlock is to perform a **release operation** and of a mutex lock to perform an **acquire operation**.
- The write by Thread 1 is said to **happen before** the read by Thread 2.
Recall, a release makes previous writes visible to other threads.

A release orders memory accesses so that no preceding write may be moved to after the release by the compiler or hardware.

An acquire makes writes by other threads that have made a release visible.

An acquire orders memory accesses so that no subsequent read or write may be moved to before the acquire.

A consume is similar to an acquire but it lets unrelated reads be moved by the compiler to before the consume. In addition it can be implemented faster on some machines including POWER.

Release/acquire is different from unlock/lock but unlock/lock perform release/acquire in addition to keeping track of the lock value.
Two expression evaluations conflict if they access the same memory location and at least one of them modifies that location.

For example:

```c
// Thread 1
a = b + c;

// Thread 2
d = a + 1;
```

An assignment is an expression so the above statements conflict.

By evaluation is meant that the expressions are computed at runtime.

Conflicting evaluations are necessary in parallel programs unless each thread can work exclusively on its own data.

Conflicting evaluations become a big problem if they are not ordered through the happens before relation.
- It is the unlock/lock pair which guarantees that the write happens before the read and that the data becomes visible to the other thread.
- Since there is no synchronization in the previous slide there is a data race, obviously.
- An unlock on \( m \) synchronizes with a lock on \( m \).
- Recall that mutex unlock/lock is one of three main kinds of synchronizations which orders two variable accesses.
- The other two are:
  - Release on an atomic object \( M \) followed by acquire or consume on \( M \)
  - Memory fences plus write and read of an atomic object \( M \)
An essential property of parallel C programs is that we have ordered all memory accesses through the happens before relation which will be explained in detail soon.

If you use mutexes to order the accesses, you will be safe.

Atomic objects and fences are intended for use when:
- Mutexes don’t give sufficient performance.
- You implement other high level synchronization primitives.

We will next explain the happens before relation in detail.

Then we will show concrete examples of using atomic objects including implementation details for POWER processors.
The most common sequence point in C/C++ is semicolon. Others include:
- Function call
- Comma operator
- After evaluating the left operand of ?:, && and ||.

Below at L, the assignment to and use of v are not sequenced.

```c
int u = 1, v = 2, w;
L: w = (v = u + 3) + v * 4;
```

It is legal C but the value of w can become either 12 or 20. It is due to it is unspecified which operand of + is evaluated first.

In the following w becomes 16 since comma is a sequence point.

```c
w = (v = u + 3), v * 4;
```
Memory Fences

- These are also called **memory barriers** and are used frequently in the Linux kernel.
- A memory fence is one of:
  - release fence
  - acquire fence
  - both release and acquire fence
- A fence has no memory location operand.
- Two threads $T_1$ and $T_2$ synchronize using a fence if $T_1$ writes to an atomic object $M$ after the release fence which is read by $T_2$ before the acquire fence.
An Example

```c
#define Atomic int m;

int u;

// T1 T2
u = 1;
atomic_thread_fence(memory_order_release); // A
atomic_store_explicit(&m, 1, memory_order_relaxed); // X

while (atomic_load_explicit(&m, memory_order_relaxed) == 0) // Y
    ;
atomic_thread_fence(memory_order_acquire); // B
printf("u = %d\n", u);
```

- Accesses to atomic objects do not produce data races.
- Simply running the two threads does not order them in any way.
- Only if Thread 2 reads m after the write then it knows that the value of u is correct.
- The modification of u happens before the read of u.
A release fence $A$ synchronizes with an acquire fence $B$ if there exist atomic operations $X$ and $Y$ which operate on an atomic object $M$ and

- $X$ is sequenced after $A$,
- $Y$ is sequenced before $B$,
- $Y$ reads a value written by $X$ or the hypothetical release sequence headed by $X$ if it were a release.

\[
\begin{align*}
u &= 1 \\
A: & \text{ release fence} \\
X: & \text{ atomic store } M \text{ relaxed} \\
Y: & \text{ atomic load } M \text{ relaxed} \\
B: & \text{ acquire fence} \\
\text{print } u
\end{align*}
\]

One of $A$ and $B$ can be replaced with an atomic operation, which eliminates the need for either $X$ or $Y$.

- $A$ and $X$ can be replaced with a release operation on $M$.
- $Y$ and $B$ can be replaced with an acquire operation on $M$. 
Wrong Synchronize-with using Fences (1)

Thread 1
u = 1
X: atomic store M relaxed
A: release fence

Thread 2
Y: atomic load M relaxed
B: acquire fence
print u

- Assignment to u and X can be reordered
- Thread 2 can read M and perform B
- Thread 2 can then read u
- Then the invalidation of u can reach Thread 2
Wrong Synchronize-with using Fences (2)

Thread 1
u = 1
A: release fence
X: atomic store M relaxed

Thread 2
B: acquire fence
Y: atomic load M relaxed
print u

- In Thread 2 reading u and Y can be reordered
- B can have been executed before A
- In this and the previous slide the fence is not between the other operations so they can be reordered.
- The Y operation should not read just any value but a value which communicates the fact that u has been updated.
Dependences are intra-thread (within one thread only).

There are two kinds of dependences:
- Data dependence — due to writing and reading a memory location.
- Control dependence — due to branches such as if-statements.

*Only* data dependences are considered in C11.

As we will see, this can lead to unexpected results.

When we talk about ”dependences” from now on, we only mean data dependences.
Intra-thread = within one thread

Consider two expression evaluations $A$ and $B$ made by one thread.

When we say that $A$ carries a dependency to $B$ it means $A$ must be evaluated before $B$:

$$v = u + 1;\quad //\ A$$
$$w = v * 2;\quad //\ B$$

The order of $A$ and $B$ should not be changed!

Both compilers and hardware must preserve this order — and they do.

$A$ is sequenced before $B$ since there is a sequence point between $A$ and $B$. 

---

Jonas Skeppstedt (jonasskeppstedt.net)
Definition of Dependency in C11

- A carries a dependency to B if
  - the value of A is used as an operand of B (except the left operand of `?:`, `&` or `||`), or
  - A writes a scalar object (pointer or arithmetic variable) or a bitfield in memory location M and B reads from M the value written by A, and A is sequenced before B
  - For some evaluation X, A carries a dependence to X and X carries a dependency to B.

- Carries a dependency is intra-thread.
A release sequence is a maximal sequence of modifications of \( M \) headed by a release operation on \( M \) and performed by that thread or by other threads performing atomic read-modify-write accesses on \( M \).

Recall, a consume operation is like an acquire except it allows more optimizations on some machines, e.g. on the POWER it becomes an ordinary load instruction.

An evaluation \( A \) in one thread is dependency ordered before an evaluation \( B \) in another thread if:

- \( A \) performs a release operation on an atomic object \( M \) and \( B \) performs a consume operation on \( M \) and reads a value written by any side effect of \( A \) in the release sequence of \( A \), or
- for some evaluation \( X \), \( A \) is dependency ordered before \( X \) and \( X \) carries a dependency to \( B \).
Inter-Thread Happens Before

An evaluation $A$ **inter-thread happens before** an evaluation $B$ if:
- $A$ synchronizes with $B$, or
- $A$ is dependency ordered before $B$, or
- for some evaluation $X$:
  - $A$ synchronizes with $X$ and $X$ is sequenced before $B$, or
  - $A$ is sequenced before $X$ and $X$ synchronizes with $B$, or
  - $A$ inter-thread happens before $X$ and $X$ inter-thread happens before $B$.

An evaluation $A$ **happens before** an evaluation $B$ if:
- $A$ is sequenced before $B$ (intra-thread), or
- $A$ is inter-thread happens before $B$.

C11, Section 5.1.2.4 paragraph 25:

*The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior.*
Motivation for Dependency Ordering

- Improved performance!
- In programs (such as the Linux kernel) with important data structures which are rarely modified and very frequently read there exist faster solutions than using full release/acquire synchronization on modern architectures.
- In this sense modern architectures include POWER, MIPS and ARM.
- For other architectures including x86, optimizing compilers can make better optimizations if dependency ordering is used rather than release/acquire, as we will see below.
RCU is an alternative to readers-writers locks with the following essential functionality:

- Pointers to data structures are protected with RCU.
- Readers use read-side critical sections marked by enter/exit calls.
- When a reader is in such a section a writer may not modify the data structure but instead modifies a copy of it.
- When the last reader has left, the original data structure is updated.

RCU is heavily used in the Linux kernel.

RCU was one part of the SCO vs IBM lawsuit.

From linux-2.6.37.1/kernel/signal.c

```c
/*
 * Protect access to @t credentials. This can go away when all
 * callers hold rcu read lock.
 */
rcu_read_lock();
user = get_uid(__task_cred(t)->user);
atomic_inc(&user->sigpending);
rcu_read_unlock();
```
Consider the following example code (also originally from Linux)

```c
list_t* head;
int b;

void insert(int a)
{
    list_t* p = kmalloc(sizeof *p, GFP_KERNEL);
    spin_lock(&m);
    p->a = 1;
    p->next = head;
    rcu_assign_pointer(head, p);
    spin_unlock(&m);
}

void first(void)
{
    list_t* q = rcu_dereference(head);
    return q->a + b;
}
```

The use of `rcu_dereference()` must be done within an `rcu_read_lock() / rcu_read_unlock()` — not seen in this example.
int b;

void first(void)
{
    list_t* q = rcu_dereference(head);
    return q->a + b;
}

- Before the proposal for dependency ordering through release/consume, the then current draft would require the use of an acquire operation in the `rcu_dereference`.
- Doing so prevents the compiler from loading `b` before the execution of the `rcu_dereference`.
- A standard for a high performance language must permit extensive compiler optimization and efficient execution on modern machines.
- C11 succeeds with this — also for multicores.