Threads in the new ISO C Standard from 2011

Contents of Lecture 6

- Atomic objects (recall, object in ISO C terminology = "data")
- Memory consistency model for non-atomic objects
- Synchronize with operation
- Dependency ordered memory accesses
- Happens before relation and data races
- Motivation from the Linux kernel for dependency order
- Implementation on POWER
- Threads API in C11 (very similar to Pthreads)

C11

- Current ISO C Standard is C11 from December 2011.
- See `http://www.open-std.org/jtc1/sc22/wg14`
- C11 contains various news but we will focus on three in this course:
  - Section 5.1.2.4 Multi-threaded executions and data races
  - `<threads.h>` — similar to Pthreads
  - `<stdatomic.h>` — types, operators, and functions for atomic objects

Atomic Objects

- Atomic objects are new in C11 and similar to Java volatile variables.
- Atomic objects can safely be accessed without locking:

```
_Atomic int counter = ATOMIC_VAR_INIT(0);

Thread 1                    Thread 2
counter++;                  counter++;
```

- Of course, you may need locks for other reasons to protect your data.
- Atomic objects have a global modification order and all threads see the same modification order.
- There is no total modification order of all atomic objects — different threads can see the stores to different atomic objects in different orders.

Syntax for Declaring Atomic Objects

- A new type qualifier: `_Atomic`.
- The other type qualifiers are: `const`, `volatile`, `restrict`.
- Examples:

```
_Atomic int                     a; // atomic int
int* _Atomic                    b; // atomic pointer
_Atomic int*                    c; // pointer to atomic int
_Atomic int* _Atomic            c; // atomic pointer to atomic int
_Atomic struct { int d; }       e; // atomic struct
```

- In C++ it's written as `atomic<int> a`.
- For easier porting C allows: `_Atomic` (*type-name*)

Using Atomic Objects

- Members of an atomic struct/union may not be accessed individually.
- The whole struct must first be copied to a non-atomic variable of compatible type.
- The ++, --, and compound assignment operators (e.g. +=) are atomic read-modify-write operations.
- The size of atomic and non-atomic compatible types is typically different as well as the alignment requirements.
- The memory ordering when using these operators is sequential consistency, which means costly memory fences are used.
- We will see functions for performing the same atomic operations which use relaxed memory ordering and may be preferable.
- Recall that alignment requirement refers to that for example a four byte `int` must be given an address that is a multiple of four etc.

Standard Atomic Data Types in <stdatomic.h>

- <stdatomic.h> defines basic atomic types including

atomic_char	atomic_schar	atomic_uchar
atomic_short	atomic_sshort	atomic_ushort
atomic_int	atomic_sint	atomic_uint
atomic_long	atomic_slong	atomic_ulong
atomic_llong	atomic_sllong	atomic_ullong
atomic_wchar_t	atomic_intptr_t	atomic_uintptr_t
atomic_address	atomic_bool	atomic_flag

- atomic_flag is a lock-free struct.
- The other types might be implemented with locks.
- An atomic flag can be initialized with ATOMIC_FLAG_INIT.

Lock Free Property

- To know whether the other basic atomic types are lock free, the following macros can be evaluated:

```
ATOMIC_CHAR_LOCK_FREE
ATOMIC_CHAR16_T_LOCK_FREE
ATOMIC_CHAR32_T_LOCK_FREE
ATOMIC_WCHAR_T_LOCK_FREE
ATOMIC_SHORT_LOCK_FREE
ATOMIC_INT_LOCK_FREE
ATOMIC_LONG_LOCK_FREE
ATOMIC_LLONG_LOCK_FREE
ATOMIC_ADDRESS_LOCK_FREE
```

- If for example `ATOMIC_INT_LOCK_FREE` is true then both `_Atomic signed int` and `_Atomic unsigned int` are lock-free.

Initializing an Atomic Object

- The initialization itself is **not** atomic!
- Either use `ATOMIC_VAR_INIT(value)`, or
- `void atomic_init(volatile A* ptr, C value);`

Memory Order for Atomic Operations

- The enumerated type `memory_order` contains the enumerators

```
memory_order_relaxed
memory_order_consume
memory_order_acquire
memory_order_release
memory_order_acq_rel
memory_order_seq_cst
```

- They are used with the functions operating on atomic objects described next.
- For example:

```
x = atomic_load_explicit(&a, memory_order_relaxed);
y = atomic_load(&b);
```

- Without `_explicit`, `memory_order_seq_cst` is used.
- As we will see in detail the code becomes faster and more confusing with `memory_order_relaxed`. Crashing fast is better.

Question

- Consider the code where all variables initially are zero.

```
// Thread 1
x = atomic_load_explicit(&b, memory_order_relaxed);
atomic_store_explicit(&a, x, memory_order_relaxed);

// Thread 2
y = atomic_load_explicit(&a, memory_order_relaxed);
atomic_store_explicit(&b, 42, memory_order_relaxed);
```

- Can both `x` and `y` become 42?

Answer: Yes

- The code may execute in the following order:

```
atomic_store_explicit(&b, 42, memory_order_relaxed);        // Thread 2
x = atomic_load_explicit(&b, memory_order_relaxed);         // Thread 1
atomic_store_explicit(&a, x, memory_order_relaxed);         // Thread 1
y = atomic_load_explicit(&a, memory_order_relaxed);         // Thread 2
```

- With relaxed memory ordering and no dependency the store can be reordered and execute first.
- There is a dependency through the variable `x` between the accesses by Thread 1 — so they may not be reordered, luckily.
- We will see more about dependences below.

Atomic Exchange Macros

- 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:

```
if (*ptr == *expected)
        *ptr = value;
else
        *expected = *ptr;
```

Atomic Compare and Exchange Function Prototypes

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

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

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

- ```
  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.

Weak Compare and Exchange Functions

- 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 ide and ta 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.

Using the Weak Compare and Exchange Functions

```
_Atomic int     a;
int             exp;
int             value;

exp = atomic_load(&a);
do
        value = f(exp);
while (!atomic_compare_exchange_weak(&a, &exp, value));
```

Atomic Fetch and Modify Functions

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

computation	C operator	function
addition	`+`	`atomic_fetch_add_explicit`
subtraction	`-`	`atomic_fetch_sub_explicit`
or	`|`	`atomic_fetch_or_explicit`
xor	`^`	`atomic_fetch_xor_explicit`
and	`&`	`atomic_fetch_and_explicit`

- 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.

Test and Set Atomic Flag

- The type `atomic_flag` is the atomic type for the classic test-and-set operation.

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

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.

Clear Atomic Flag

```
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`

The C11 Memory Consistency Model for Non-Atomic Objects

- 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`

```
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.

Release and Acquire, and Consume

- 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/aqcuire in addition to keeping track of the lock value.

Conflicting Expression Evaluations

- Two expression evaluations **conflict** if they access the same memory location and at least one of them modifies that location.
- For example:

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

```
// Thread 2

d = a + 1;
```

- An assignment is an expression so the code above 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.

Happens Before and Data Races

- We will look at the details of the happens before relation and what it means in terms of executing C code at the machine instruction level below, after introducing happens before informally.
- 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.
- There are additional important details for atomic objects and fences which we will see later.

Summary so far

- 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.

Sequenced Before

- 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.

```
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.

```
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.
- Details on the following slides.

An Example

```
_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`.

Synchronize-with using Fences

- 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.

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

```
Y: atomic load M relaxed
B: acquire fence
print u
```

- 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                          Thread 2
u = 1
X: atomic store M relaxed
A: release fence

                                  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
- Thread 2 cannot know it has actually read a 1 in M
- B can have been executed before A

Dependences

- 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.

Dependences between Intra-Thread Evaluations

- 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;        // A
w = v * 2;        // 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.

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.

Dependency-Ordered

- 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 (mutex unlock/lock), or
  - A is dependency ordered before B (release/consume), 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.

Read-Copy Update

- 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

```
/*
 * 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();
```

From the Linux Kernel Documentation

```
rcu_dereference()

typeof(p) rcu_dereference(p);

Like rcu_assign_pointer(), rcu_dereference() must be implemented
as a macro.

The reader uses rcu_dereference() to fetch an RCU-protected
pointer, which returns a value that may then be safely
dereferenced.  Note that rcu_deference() does not actually
dereference the pointer, instead, it protects the pointer for
later dereferencing.  It also executes any needed memory-barrier
instructions for a given CPU architecture.  Currently, only Alpha
needs memory barriers within rcu_dereference() -- on other CPUs,
it compiles to nothing, not even a compiler directive.
```

A Simplified Example

- Consider the following example code (also originally from Linux)

```
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.

A More Detailed Look

```
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.

More Details About Dependences

- Recall the intra-thread A carries a dependency to B.
- The compiler becomes responsible for not optimizing away such dependences.
- This might sound trivial but is not. See below.
- A dependency is started with a consume operation:

```
p = atomic_load(q, memory_order_consume);
a = *p + 1;
```

- A root of a dependency tree is created with the consume, and the memory read in `*p` becomes ordered, as we expect.
- The compiler or hardware is not allowed to move the `*p` to before the first line which would probably not make much sense, anyway.
- Behaviour is as we want and expect.

Constant Propagation Example 1(3)

- Consider the following code and assume the compiler has deduced `size` is one:

```
i = atomic_load(q, memory_order_consume);
a = b[i % size];
```

- What happens if the compiler transforms the code to the following?

```
i = atomic_load(q, memory_order_consume);
a = *b;
```

Constant Propagation Example 2(3)

- There is now **no dependence** between the two statements!

```
i = atomic_load(q, memory_order_consume);
a = *b;
```

- Without a dependence the reads are not ordered.
- This is not what the programmer expected!!!
- Therefore, optimizing C compilers must analyse all dependences **before** any code transformation and preserve them.
- What does "preserve" mean, then?

Constant Propagation Example 3(3)

- Preserving the data dependency means letting the hardware know about them.
- That can be done in different ways for different processors.
- One way is to insert instructions so that there will be a chain of dependences for the processor pipeline to see.
- Compilers thus must preserve such dependences in the complete program!
- Writing optimizing compilers C all of a sudden became twice as interesting.
- There is, however, a very simple first version implementation: treat all `memory_order_consume` as `memory_order_acquire` which automatically will order the memory accesses.

One More Data Dependency

```
a = atomic_load(p, memory_order_consume);
atomic_store(q, a, memory_order_relaxed);
atomic_store(r, b, memory_order_relaxed);
```

- Since there is no data dependency between the consume and the second store, they are not ordered.
- As programmers we need to be very careful about what we expect to be ordered!

More Details Next

- Implementation on a specific architecture: POWER.
- Illustrations of what actually is ordered for several situations.
- We will begin with the POWER synchronization instructions.
- Approximate clock counts below are not fixed but depends on what is happening in the machine at the moment.
- Numbers from Christoph von Praun: *"Deconstructing Redundant Memory Synchronization"* (while at IBM Research).

mnemonic	name	POWER4 cycles	POWER5 cycles
`ldarw/stwxc.`	load and reserve / store conditional	80	75
`hwsync` or `sync`	heavy weight sync	140	50
`lwsync`	light weight sync	110	25
`isync`	instruction sync	30	10
`eieio`	enforce inorder execution of I/O	not measured	not measured

- We want to use functions which use the less costly instructions!
- Such as the consume operation which is only an ordinary load instruction on POWER!

Load and Reserve/Store Conditional Purpose

- Some processors use atomic test-and-set instructions while others use pairs of special load and store instructions.
- Load and reserve is also called load-locked and load-linked.
- In addition to POWER, it's used by ARM and MIPS.
- Test-and-set, or compare-and-swap, is used by e.g. x86.
- The purpose with load-and-reserve/store conditional is to simplify the design of the pipeline.

Load and Reserve/Store Conditional Behaviour

- The load instruction fetches data from a memory location, and makes a reservation R in memory.
- If a different processor modifies the same memory location, the reservation R is lost.
- If/when the processor with a reservation makes a conditional store, the memory location is modified only if the reservation was not lost in between.
- Therefore it's an atomic read-modify-write.
- The `stwcx.` conditional store in POWER sets a condition code to indicate whether the store succeeded or not (the dot in the mnemonic indicates that the condition code is set by an operation on POWER).

POWER Memory Barrier Instructions

- The POWER memory barrier instructions create a memory barrier with two sets of instructions:
  - the A-set with instructions a_i preceding the barrier, and
  - the B-set with instructions b_j following the barrier.
- Depending on which barrier instruction is used, some b_j instructions may be reordered with some a_i instructions.

`hwsync` Memory Barrier for Sequential Consistency

- The *A*-set consists of all instructions preceding the `hwsync`.
- The *B*-set consists of memory access instructions following the `hwsync`.
- Except for the instruction `icbi` which invalidates an instruction cache block, no *B*-set instruction may be reordered with any *A*-set instruction.
- This is the most costly POWER synchronzation instruction.
- Not only for cached data but for any storage.
- It's used e.g. to implement sequentially consistent write on POWER:

```
stwx    r1,r2,r3
hwsync
```

`lwsync` Memory Barrier for Release Operation

- The A and B sets consist of all memory access instructions preceding and following the `lwsync`, respectively.
- Only the following pairs of a_i, b_j instructions are ordered:
  - A-side load $\rightarrow$ B-side load
  - A-side load $\rightarrow$ B-side store
  - A-side store $\rightarrow$ B-side store
- Thus, A-side store $\rightarrow$ B-side load are not ordered.
- The `dcbz` data cache block zero is counted as a store.
- It's used e.g. to implement a release:

```
stwx    r1,r2,r3          # modify shared data...
stwx    r4,r5,r6
...
stwx    r29,r30,r31       # last write in critical section
lwsync
stwx    r8,r9,r10         # set lock to free
```

`isync` Memory Barrier for Acquire Operation

- The A and B sets consist of all instructions preceding and following the `isync`, respectively.
- An instruction which cannot raise an exception in the pipeline can be allowed to complete and instructions following the `isync` therefore actually execute without order.
- Consider the sequence:

```
ldw     r1,r2,r3
isync
stw     r4,r5,r6
```

- The store may execute before the load if the load had e.g. a cache miss.
- This is not what we want and to overcome that problem we can exploit that POWER does not permit speculative execution of store instructions.

bc;isync Memory Barrier for Acquire Operation

- By inserting a conditional branch, `bc`, before the `isync` both the `isync` and `stw` become speculative until the branch outcome is known.
- We can use beq as follows:

```
ldw     r1,r2,r3 # r1 = MEMORY[r2+r3]
cmp.    r1,r1    # certainly true but the beq must
beq              # wait according to the specification
isync            # since no speculative stw is allowed.
stw     r4,r5,r6
```

- Since the store may not execute speculatively it must wait for the branch outcome.
- This memory barrier is the fastest.
- The previous two, however, can order instructions from different processors due to the `hwsync`/`lwsync` are **cumulative** — see below.

`eieio` Memory Barrier

- (unique five vowel instruction)
- Enforce in order execution of I/O.
- It only orders stores.

Cumulative Ordering

- If the memory accesses ordered by a memory barrier executed by one processor P_i also take into account memory accesses executed by other processors as described below the barrier is **cumulative**.
- By **applicable** storage accesses for a barrier is meant the storage access which are ordered by that barrier.
- Two rules:
  - The A-set also includes all applicable storage accesses made by other processors which have completed with respect to P_i before the barrier is created (by executing the barrier instruction).
  - The B-set also includes all applicable accesses made by any processor P_j after P_j has executed a load that returned a value stored by an instruction in the B-set.
- The B-set expands recursively.

Example of Cumulative Ordering

```
// int a = b = 0;
// Thread 1                Thread 2                  Thread 3
1:a = 1;
2:lwsync
                           3:x = a; // a == 1
                           4:y = b;
                           5:lwsync
B = { 2,3,5,6 }            6:b = 2;
                                                     7:x = b; // b == 2
                                                     8:lwsync
                                                     9:y = a; // a == 1

A = { }                    A = { 2,3,4 }             A = { 2,4,7 }
B += { 7,8 }               B = { 6,7,9 }             B = { 9 }
```

- Due to cumulativity of `lwsync`, it's certain that **if** Thread 2 reads 1 in access 3 **and** Thread 3 reads 2 in access 7 **then** Thread 3 will read 1 in access 9.
- Using instead `bc;isync` Thread 3 may read zero in access 9.

Implementation on POWER

- The following is based on the ISO C/C++ standardization document ISO/EIC JTC1 SC22 WG21 N2745 written by Paul E McKenney and Raul Silvera from IBM.
- There are other implementations possible.

Load

Load relaxed	`ld`
Load consume	`ld`
Load acquire	`ld; cmp; bc; isync`
Load seq cst	`hwsync;ld; cmp; bc; isync`

1. Using sequential consistency on machines with relaxed memory models makes it easier to write correct parallel code which will be slow.
2. Even if your code is safety critical (when people can die due to bugs) and performance is not an issue, you should not use SC, use a single threaded C program instead.

Store

Store relaxed	`st`
Store release	`lwsync;st`
Store seq cst	`hwsync;st`

1. It's easier to program under sequential consistency!
2. Yes, but, why would you want to use synchronization instructions before every store in a critical section since nobody should look at the data before you have left the critical section anyway?
3. Write buffering permitted by store relaxed allows the machine to pipeline all the stores in the critical section.
4. At the end of the critical section you use store release and wait for the remaining preceding stores (now possibly waiting in a write buffer) to complete (i.e. invalidate the other copies if there are any left).

Memory Fence

Acquire fence	`lwsync`
Release fence	`lwsync`
Acq rel fence	`lwsync`
Seq cst fence	`hwsync`

Acquire and Release a Spin Lock

```
        # acquire
loop:   lwarx   r6,0,r3         # load lock and reserve
        cmpw    r4,r6           # r4 is a free lock's value
        bne-    loop            # restart if locked.
        stwcx.  r5,0,r3         # try to store
        bne     loop            # restart if store failed.
        isync                   # store succeeded.
        lwzx    r7,r8,r9        # first load of shared data

        # release
        stwx    r7,r8,r10       # last store of shared data
        lwsync                  # export shared data
        stw     r4,0,r3         # unlock the lock. same r4 as above.
```

<threads.h> Header File

- C11 threads are similar to but simpler than Pthreads.
- `thrd_t` — thread type
- `once_flag` — a type for performing initializations exactly one time
- `mtx_t` — mutex type
- `cnd_t` — condition variable type
- `tss_t` — thread specific storage (not the same as `_Thread_local`)

Initialization Function

- `void call_once(once_flag* flag, void (*func)(void));`
- The flag should be initialized with:

```
once_flag flag = ONCE_FLAG_INIT;
```

- If multiple threads invoke `call_once` with the same flag, the function will only be called one time, and the others will wait until the call to `func` returns.

Thread Enumeration Constant Error Codes

- `thrd_success` — indicates an operation succeeded.
- `thrd_error` — indicates an operation failed but not why.
- `thrd_busy` — an operation failed due to a resource was already in use.
- `thrd_nomem` — an operation failed due to memory allocation failed.
- `thrd_timeout` — a timed wait operation timed out.

Mutex Options for `mtx_init` Function

- `mtx_plain` — the mutex should support none of below options.
- `mtx_recursive` — set mutex to support recursive locking.
- `mtx_timed` — set mutex to support timed wait.
- `mtx_try` — set mutex to support test and return.

Specifying Waiting Time

- The struct `xtime` contains at least the following members.
- `time_t sec;`
- `long nsec;`
- They may be declared in any order in the struct.
- This means that code which compares two char-pointers (each pointing to one of them) should not assume one is before the other.
- In `struct { int a, b; } c; int* p = &c.a; int* q = &c.b;` we know that `p < q`, since `a` is declared before `b`.

Condition Variable Functions

```
int cnd_init(cnd_t* cond);
void cnd_destroy(cnd_t* cond);
int cnd_signal(cnd_t* cond);
int cnd_broadcast(cnd_t* cond);
int cnd_wait(cnd_t* cond, mtx_t* mtx);
int cnd_timedwait(cnd_t* cond, mtx_t* mtx, const xtime* xt);
```

- These functions are all similar to the corresponding Pthreads functions, except that Pthread condition variables can have certain attributes and be statically initialized.

```
int pthread_cond_init(
        pthread_cond_t* restrict                cond,
        const pthread_condattr_t* restrict      attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Mutex Functions

```
int mtx_init(mtx_t* mtx, int type);
void mtx_destroy(mtx_t* mtx);
int mtx_lock(mtx_t* mtx);
int mtx_timedlock(mtx_t* mtx, const xtime* xt);
int mtx_trylock(mtx_t* mtx);
int mtx_unlock(mtx_t* mtx);
```

- The `type` should be one of:

  `mtx_plain`
  `mtx_timed`
  `mtx_try`
  `mtx_plain | mtx_recursive`
  `mtx_timed | mtx_recursive`
  `mtx_try | mtx_recursive`

- Only `mtx_trylock` can return `thrd_busy`.
- A prior call to `mtx_unlock` synchronizes with a successful call to a `mtx_lock` function.

Thread Functions

```
int thrd_create(thrd_t* thr, int (*func)(void*), void* arg);
void thrd_exit(int res);
int thrd_join(thrd_t thr, int* res);
int thrd_detach(thrd_t thr);
thrd_t thrd_current(void);
int thrd_equal(thrd_t u, thrd_t v);
void thrd_sleep(const xtime* xt);
void thrd_yield(void);
```

Thread Specific Storage Functions

```
int tss_create(tss_t* key, void (*dtor)(void*));
void tss_delete(tss_t key);
void* tss_get(tss_t key);
int tss_set(tss_t key, void* value);
```