5-1: Multithreading (Theory)

Artem Pavlov, TII, Abu Dhabi, 18.04.2024

Threads

- Common abstraction for independent execution of code
- Execution of threads code can be interleaved or performed in parallel
- Threads in one process work with the common address space
- Each thread usually has its own stack memory

Spawning threads

- Threads can be spawned using std::thread::spawn function
- Threads can be configured using std::thread::Builder API
- Spawned threads return JoinHandle which can be used to wait for spawned thread’s termination and retrieval of return result
- Threads can not be killed with std APIs!

Spawning scoped threads

- If spawned threads need to borrow non-static data, you can use std::thread::scope to create “scoped” threads
- Scoped threads involve two lifetimes: 'scope and 'env ('env: 'scope bound is part of the Scope type)
- The 'scope lifetime represents the lifetime of the scope itself
- The 'env lifetime represents the lifetime of whatever is borrowed by the scoped threads

Send

- Auto marker trait used for types that can be transferred across thread boundaries
- Commonly used by APIs which deal with sending data across threads
- If Send was not implemented for a type, it can be implemented manually using unsafe

Sync

- Auto marker trait for types for which it is safe to share references between threads
- In other words, a type T is Sync if and only if &T is Send
- &mut T is Sync, because mutation through &&mut T is not possible
- Counter-example: types with interior mutability Cell and RefCell are not Sync
- Atomic types (e.g. AtomicU32 and Arc) are Sync

Relation between Send and Sync

- &T is Send if and only if T is Sync
- &mut T is Send if and only if T is Send
- &T and &mut T are Sync if and only if T is Sync
- More in the Rustonomicon: https://doc.rust-lang.org/nomicon/send-and-sync.html

Thread Local Storage

- Can be created using the thread_local! Macro
- Creates static with LocalKey<T> type
- Only shared (&T) references pointing to TLS may be obtained from LocalKey
- This restriction can be worked around by using interior mutability, e.g. with Cell and RefCell

Mutex

- A mutual exclusion primitive useful for protecting shared data
- The protected data can only be accessed through the RAII guards returned from lock and try_lock
- If during mutex locking code panics, it poisons the mutex
- Common pattern is to use Arc<Mutex<T>>

RwLock

- A variation of Mutex which supports multiple read locks or exclusive write lock
- The priority policy of the lock is dependent on the underlying operating system's implementation
- Gets poisoned if code panics with acquired write lock

MPSC channels

- Multi-producer, single-consumer FIFO channels
- Defined in the std::sync::mpsc module
- std provides two variations: unbounded (mpsc::channel) and bounded (mpsc::sync_channel)
- Channels have two parts: sender and receiver
- Sender types can be cloned

Atomics

- Atomic types provide primitive shared-memory communication between threads
- Std provides the following atomic types: AtomicBool, AtomicPtr, AtomicIsize, AtomicUsize, AtomicI8, AtomicU16, AtomicU32, etc.
- Atomics can be stored in statics

Ordering of atomic operations

- Rust atomics currently follow the C++20 model
- Most atomic operations take Ordering enum
- Ordering has the following variants:
  Relaxed, Release, Acquire, AcqRel, SeqCst
- Read more about atomics in the “Rust Atomics and Locks” book: https://marabos.nl/atomics/

Case study: rayon

- rayon is a data-parallelism library for Rust.
- Makes it easy to convert a sequential computation into a parallel one
- Ergonomic: in adaptor-based code you often need only replace iter with par_iter
- Uses worker stealing thread pool to efficiently implement parallel processing of data
- Example “Computing prime numbers in parallel”: https://wannesmalfait.github.io/blog/2024/parallel-primes/

Questions?