Bringing Per-CPU Variables to Rust for Linux

Mitchell Levy

PerCPU Summary

- Locking and atomics are expensive
  - Atomics are only (really) available for machine intrinsics
- Avoid data races by giving each CPU its own variable
- Only lock when we want to aggregate variables across CPUs
- Good for performance critical code!

Caveats

- This talk is focused on x86_64
  - Any arch-specific tricks won't work on other platforms, of course
- The Rust-for-Linux project moves very quickly
- This work is very much ongoing!
  - Goal: Get a flavor for "doing Rust-for-Linux work"
  - Non-Goal: How to use the per-CPU API from Rust or from C

Per-CPU Implementation

- Each CPU is assigned a Per-CPU area in memory
  - On x86, the location of this area is put in the `gs` segment register
- Want to add 1 to my_percpu_var?
  - `addq %gs:[my_percpu_var] 1`

Per-CPU Implementation

- Who uses segmented addressing in 2025??
  - Not your compiler! (Mostly)
  - Per-CPU functions are macros that generate inline assembly
  - Exactly as horrifying as you think
- None of these operations are atomic (would defeat the point)
  - Need to make sure we don't switch CPUs at inconvenient points

Per-CPU Implementation

```
#define percpu_to_op(size, qual, op, _var, _val)                        \
do {                                                                    \
        __pcpu_type_##size pto_val__ = __pcpu_cast_##size(_val);        \
        if (0) {                                                        \
                typeof(_var) pto_tmp__;                                 \
                pto_tmp__ = (_val);                                     \
                (void)pto_tmp__;                                        \
        }                                                               \
        asm qual(__pcpu_op2_##size(op, "%[val]", __percpu_arg([var]))  \
            : [var] "+m" (__my_cpu_var(_var))                          \
            : [val] __pcpu_reg_imm_##size(pto_val__));                 \
} while (0)

#define percpu_unary_op(size, qual, op, _var)                           \
({                                                                      \
        asm qual (__pcpu_op1_##size(op, __percpu_arg([var]))           \
            : [var] "+m" (__my_cpu_var(_var)));                        \
})
```

Per-CPU in Rust

- Can't reuse a lot of the C infrastructure
  - We'd need a helper function for each combination of supported asm instruction and operand width
  - This would also add function call overhead

Static Per-CPU Variables

Static Per-CPU Variables

- The actual symbol declared as a Per-CPU variable is "fake"
  - Need to prevent users from reading from it directly
- Each Per-CPU variable's address is effectively an offset into this area
  - How? Linker magic

Static Per-CPU Variables in C

```c
DEFINE_PER_CPU(int, x);
int z;

z = this_cpu_read(x);
// mov ax, gs:[x]
```

Per-CPU in Rust

- Rust aliasing rules
  - Cannot ever have a `&mut T` that aliases a `&T` or another `&mut T`
  - Makes manually moving between pointers and references tricky
  - All of this is just juggling pointers around! A PerCPU "reference" is the current CPU's PerCPU area + offset of the variable
  - Need to prevent users from doing this twice (hopefully in a way the compiler can enforce, or with a minimum of `unsafe` code)

Current API

```
define_per_cpu!(PERCPU: u64 = 0);
// expands to:
static __INIT_PERCPU: u64 = 0;
#[link_section = ".data..percpu"]
static PERCPU: StaticPerCpuSymbol<u64> = unsafe {
  transmute::<u64, StaticPerCpuSymbol<u64>>(__INIT_PERCPU)
};
```

Current API

```
pub struct PerCpuRef<T> {
  offset: usize, guard: CpuGuard, /* others */
}

// PerCpuRef<T> behaves like a &T or a &mut T
```

Current API

```
define_per_cpu!(PERCPU: u64 = 0);
let pcpu_ref = unsafe {
  // unsafe_get_per_cpu_ref!(PERCPU, CpuGuard::new())
  // expands to:
  let off = ptr::addr_of!(PERCPU);
  PerCpuRef::new(off, CpuGuard::new()) // unsafe fn
}
```

Dynamic Per-CPU Variables

Dynamic Per-CPU Variables

- In C you call `alloc_percpu` and it gives you a per-CPU pointer
  - A per-CPU pointer is just an offset, analogous to the address of a static per-CPU symbol
  - Of course, you also need the size and alignment of the type
- We can just call that function from Rust!

The `PerCpuAllocation<T>` API

- We need a way to store the result of the call to `alloc_percpu`
- Why not just use `PerCpuRef<T>`?
  - When the pointer from `alloc_percpu` falls out of scope of all users, we want to free it
  - We *don't* want this to happen if a pointer to a statically-allocated variable falls out of scope
  - We *don't* want this to happen if another CPU is still using the allocation
  - We *don't* want any uses of the allocation to live longer than the allocation itself

The `PerCpuAllocation<T>` API

```
struct PerCpuAllocation<T> { offset: usize, /* … */ }
impl<T> Drop for PerCpuAllocation<T> {
  fn drop(&mut self) {
    unsafe { free_percpu(self.offset as *mut c_void) }
  }
}
```

Connecting the Dots

```rust
pub struct PerCpu<T> {
  alloc: Arc<PerCpuAllocation<T>>
}
impl<T> PerCpu<T> {
  pub fn get(&mut self, guard: CpuGuard) -> PerCpuRef<T> {
    unsafe { PerCpuRef::new(self.alloc.offset, guard) }
  }
}
```

Problem!

- What if we do something like:

```
let mut num: PerCpu<u32> = PerCpu::new().unwrap();
let mut num_ref: PerCpuRef<u32> = num.get(CpuGuard::new);
drop(num);
*num_ref = 1;
```

The Solution: Lifetimes

```
pub struct PerCpuRef<'a, T> {
  offset: usize, guard: CpuGuard, /* … */
}
impl<'a, T> PerCpuRef<'a, T> {
  pub unsafe fn new<'b>(offset: usize, guard: CpuGuard)
      -> PerCpuRef<'b, T> {
    PerCpuRef { offset, guard, /* … */ }
  }
}
```

The Solution: Lifetimes

```
impl<T> PerCpu<T> {
  pub fn get(&'a mut self, guard: CpuGuard) {
    unsafe {
      PerCpuRef::new::<'a>(self.alloc.offset, guard)
    }
  }
}
```

Problem Solved!

```
error[E0505]: cannot move out of `num` because it is borrowed
  --> lib/percpu_test_rust.rs:83:14
   |
81 |         let mut num: PerCpu<u32> = PerCpu::new().unwrap();
   |             ------- binding `num` declared here
82 |         let mut num_ref: PerCpuRef<u32> = num.get(CpuGuard::new());
   |                                           --- borrow of `num` occurs here
83 |         drop(num);
   |              ^^^ move out of `num` occurs here
84 |         *num_ref += 1;
   |       ------- borrow later used here
```

What About Static Variables?

- We still want to use the `PerCpuRef` type for statically-allocated variables
- What lifetime should they use?
  - Rust has a special `'static` lifetime
  - Essentially an unbounded lifetime

Remaining Work

- Lots of optimizations for numeric types
  - Rather than read/add/writeback numeric operations, just use a single add/sub/etc instruction with gs-relative memory operands
- Needs to work on ARM64 (and other architectures)
- Plenty of bugs waiting to be found

GitHub

- [Issues · Rust-for-Linux/linux]