课程设计中期报告

coro-core 支持协程的smp内核 (with c++20)

傅子轩
2020010742

2022/11/5

目标

- 希望在内核的部分或全部组件内引入协程，长耗时任务或外部IO使用异步调用，使逻辑更加清晰并提升性能，在高并发下减少内存使用
- 对于用户空间提供异步系统调用接口。
- 以uCore-SMP（陶天骅）作为参考，使用C++风格重写。

设计

- 每个cpu上有一个进程调度器，共享一个进程队列。统一调度用户进程和内核进程。使用时钟中断切换任务。协程调度器本身作为一个内核进程受进程调度器调度。每个cpu上有绑定在其上的一个协程调度器
- 。
- syscall来临时，对于长时间阻塞的任务，仅仅向协程调度器添加一个协程，而后立刻返回一个handle。对于大量的同类型请求，希望支持批量请求（减少特权级切换）

缺陷

- 引入协程从性能来说实际上引入了额外开销：协程创建时需要进行动态内存分配，切换时需要有函数调用，局部性变差等。
- 因此对于优化后的低并发的任务来说，引入协程反而会使性能变差。
- 另外如果希望使用C++风格来写整个内核，也容易由于疏忽引入一些不易察觉的开销。
- 许多C++的特性不可用（RTTI, 异常）
- 容易出现编译器内部错误

优势

- 更优雅的异步代码风格，可以避免所有的callback

```
int do_request1() {
    device->request(..., request_done_callback1);
}

int request_done_callback1(int good) {
    if(good) {
        device->request(..., request_done_callback2);
    } else {
        // error handling
        device->request(..., request_done_callback2);
    }

}
```

```
int request_done_callback2(int good) {
    if(good) {
        device->request(..., request_done_callback3);
    } else {
        // error handling
        device->request(..., request_done_callback3);
    }
}

int request_done_callback3(int good) {
    if(good) {

    } else {
        // error handling
    }
}
```

协程版本

```
task<int> async_do_request() {
    auto result1 = co_await device->request(...);
    if(!result1){
        // error handling
    }
    auto result2 = co_await device->request(...);
    if(!result2){
        // error handling
    }
    auto result3 = co_await device->request(...);
    co_return result3;
}
```

协程版本（统一错误处理）

```
task<int> __async_do_request() {
    auto result1 = co_await device->request(...);
    auto result2 = co_await device->request(...);
    auto result3 = co_await device->request(...);
    co_return result3;
}

task<void> async_do_request() {
    auto result = co_await __async_do_request();
    if(!result){
        // error handling
    }
}
```

优势

- 对于高并发的情况，曾经的做法是多线程，这样会为每一个线程预分配一个内核栈(2kb/4kb)，因此过多线程对于内存的开销是巨大的。但多数时候线程都在等待内核完成。并没有做其他有意义的工作。
- 在内核引入协程后，无需为每个线程分配栈。只需要为每个硬件线程分配一个栈，在trap的时候临时使用，对于syscall实际的请求，在trap时创建协程并分发到协程调度器。

优势

- C++的析构函数和移动语义可以避免许多由于疏忽引起的错误，比如忘记put inode等。

- 虽然没有C++本身的异常支持，但可以在协程中引入类似异常的支持。协程在暂停时可以安全地被销毁（按预期调用所有的析构函数）。因此在调用链的底层出错时，可以简单地将异常直接传递给最上层，避免了错误码的反复判断和返回。

- 内核线程也可以被安全地销毁了

进展

- 自底向上地，实现（重写）了：
- cpu(中断控制), spinlock, page_allocator, heap_allocator
- trap, process(kernel/user), process_scheduler
- 协程框架和协程调度器，wait_queue(协程)
- 抽象inode(协程), logger(协程),
- 抽象device/block_device(协程), virtio_blk_driver(协程), block_buffer(协程)

协程示例

- 其中的每一步都不是阻塞的（会主动让出控制权）
- 但代码逻辑与阻塞调用一致
- 注意到其中没有做任何直接的异常处理，所有的错误会统一由上层处理。

```
task<void> test_disk_write(
        int block_no, uint8* buf, int len) {

    // get buffer reference
    auto buffer_ref = co_await
        kernel_block_buffer.get_node(
                virtio_disk_id, block_no);
    auto& buffer = *buffer_ref;

    // try to get buffer for write
    std::optional<uint8*> data_ptr =
        co_await buffer.get_for_write();
    uint8* data = *data_ptr;

    for(int i = 0; i < len; i++) {
        data[i] = buf[i];
    }

    co_return task_ok;
    // ~buffer() call buffer.put() automatically
}
```

杂项进展

- 修复了一个SMP相关的bug：由于支持抢占，非原子的获取中断状态和设置中断状态可能导致错误

```
void cpu::push_off(){
    int old = intr_get();

    intr_off();
    if (noff == 0) {
        base_interrupt_status = old;
    }
    noff += 1;
}
```

```
static int local_irq_save(){
    int old = rc_sstatus(SSTATUS_SIE);
    return old;
}

static void local_irq_restore(int old){
    s_sstatus(old & SSTATUS_SIE);
}

static inline uint64 rc_sstatus(uint64 val){
    unsigned long __v = (unsigned long)(val);
    __asm__ __volatile__ ("csrrc %0, sstatus, %1"
                          : "=r" (__v) : "rK" (__v)
                          : "memory");
    return __v;
}
```

杂项进展

- 在gdb中，可以在内核出现异常时进行backtrace

TODO

- 异步文件系统
- 异步系统调用（主要是读写文件）
- 协程优先级调度

- 可能的目标（如果上面进展顺利）：
- 异步网络驱动
- 用户协程统一调度
- 缓存友好的协程调度策略
- 缺页（异步）处理

谢谢

- 仓库链接： fuzx20/coro-core