操作系统课程设计期末报告

傅子轩 2020010742

Coro-core

参考实现

2021春季学期：陶天骅的操作系统课程设计 `uCore-SMP`： [源代码仓库和文档]

已经实现的功能

- 对大部分上层模块用C++的风格(RAII, template)进行了重写。
- 固定大小内存分配器（8,16,32,64,128,256,512字节）
- 协程内部引入了类似异常的机制 `co_return task_fail`
- 协程风格的块设备读写
- 一个简单的协程风格的文件系统
- `inode/dentry` 缓存
- `debug` 模式下对协程栈和线程栈的 `backtrace`，使用 `gdb` 可以在内核异常处获得完整的调试信息

```
os/trap/trap.cc:112: LoadPageFault in kernel: 0x000000000000000d, stval = 0x0000000000000100 sepc = 0x0000000080202250
backtrace:
    test_coro2(test_coro2(device_id_t)::_Z10test_coro211device_id_t.Frame*) [clone .actor] at /mnt/ucore/ccore/os/test_nfs.cc:24
    test_coro(test_coro(void*)::_Z9test_coroPv.Frame*) [clone .actor] at /mnt/ucore/ccore/os/test_nfs.cc:34
    test_nfs_coro_unit(test_nfs_coro_unit(device_id_t)::_Z18test_nfs_coro_unit11device_id_t.Frame*) [clone .actor] at /mnt/ucore/ccore/os/test_nfs.cc:111
    test_nfs_coro(test_nfs_coro(device_id_t)::_Z13test_nfs_coro11device_id_t.Frame*) [clone .actor] at /mnt/ucore/ccore/os/test_nfs.cc:382
    __task_executor(__task_executor(promise<void>*)::_Z15__task_executorP7promiseIvE.Frame*) [clone .actor] at /mnt/ucore/ccore/os/coroutine.cc:53
backtrace stack:
    _panic(char const*, char const*, int) at /mnt/ucore/ccore/os/utils/panic.cc:36
    kernel_exception_handler(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long) at /mnt/ucore/ccore/os/arch/riscv.h:300
    virtio_disk* device::get<virtio_disk>(device_id_t) at /mnt/ucore/ccore/os/device/device.h:53
    kernel_ret at /mnt/ucore/ccore/os/trap/kernelvec.S:74
    task_base::resume() at /mnt/ucore/ccore/os/coroutine.cc:43
    task_scheduler::start() at /mnt/ucore/ccore/os/coroutine.cc:165 (discriminator 2)
    task_scheduler_run(void*) at /mnt/ucore/ccore/os/init.cc:142
    kernel_process::__kernel_function_caller(void (*)(void*), void*) at /mnt/ucore/ccore/os/arch/riscv.h:276
    kernel_process::__kernel_function_caller(void (*)(void*), void*) at /mnt/ucore/ccore/os/proc/process.cc:385
backtrace stack done
os/trap/trap.cc:129: panic: kernel exception
```

- 用户态elf格式文件加载执行（未完成 `syscall`）

已实现功能的demo

因为没有完整实现用户态，因此实现了一个运行在内核态下的简单的文件系统shell

`mkfs`

```
nfs> help
Commands: ls, mkfs, mount, unmount, cat, append, write, touch, mkdir, unlink, link, cd, help, exit
nfs> ls
fs not mounted
nfs> mkfs
[5240220] [DEBUG] [C] [2] nfs::make_fs: inode_table: 0x0000000087e4a400
[5243047] [DEBUG] [C] [3] nfs::make_fs: root_inode: 0x0000000087e4a600
[5243477] [DEBUG] [C] [3] nfs::make_fs: waiting for unmount
[5247906] [DEBUG] [C] [1] nfs: destroy block buffer
[5251472] [DEBUG] [C] [1] nfs: destroy block buffer done
mkfs success
nfs> mount
[14203403] [DEBUG] [C] [3] kernel_inode_cache size:1
[14203693] [DEBUG] [C] [3] kernel_dentry_cache size:0
mount success
nfs> ls
name inode size nlinks perm type
0 entries
nfs>
```

多级目录和链接

```
nfs> ls
name inode size nlinks perm type
0 entries
nfs> mkdir dir1
nfs> cd dir1
nfs> ls
name inode size nlinks perm type
0 entries
nfs> mkdir dir2
nfs> ls
name inode size nlinks perm type
dir2 3 0 1 0 2
1 entries
nfs> cd dir2
nfs> ls
name inode size nlinks perm type
0 entries
nfs> mkdir dir3
nfs> cd dir3
nfs> touch test
nfs> ls
name inode size nlinks perm type
test 5 0 1 0 1
1 entries
nfs> append test hello\nworld
nfs> cat test
hello
world
nfs>
```

```
nfs> cd ../../..
[49667245] [DEBUG] [C] [3] current: 'dir3' 0x0000000087df7ee0, parent: 'dir2' 0x0000000087df7ef0
[49668004] [DEBUG] [C] [3] current: 'dir2' 0x0000000087df7ee0, parent: 'dir1' 0x0000000087df7ef0
[49668642] [DEBUG] [C] [3] current: 'dir1' 0x0000000087df7ee0, parent: '' 0x0000000087df7ef0
nfs> ls
name inode size nlinks perm type
dir1 2 32 1 0 2
1 entries
nfs> link dir1/dir2/dir3/test shortcut
nfs> ls
name inode size nlinks perm type
dir1 2 32 1 0 2
shortcut 5 11 2 0 1
2 entries
nfs> cat shortcut
hello
world
nfs>
```

删除文件

```
nfs> unlink dir1/dir2/dir3/test
nfs> ls
name inode size nlinks perm type
dir1 2 32 1 0 2
shortcut 5 11 2 0 1
2 entries
nfs> unlink shortcut
nfs> ls
name inode size nlinks perm type
dir1 2 32 1 0 2
1 entries
nfs>
```

大文件支持

```
nfs> test_bigfile test
nfs> ls
name inode size nlinks perm type
dir1 2 32 1 0 2
test 6 1048576 1 0 1
2 entries
nfs>
```

一些问题

- 低估了SMP和协程的调试复杂度
- 手写的协程库有一定的性能问题（见下面的分析）
- 没有完成此前设想的异步系统调用（时间因素）

协程分析

C++中典型的协程设计

粗略阅读了`cppcoro`和`libcoro`的代码

```
task<return_type> coro_function(arg_type args) {
    do_something();

    auto ret = co_await coro_function2();

    co_return ret;
}
```

当编译器识别到函数体内有`co_return`关键字时，会把它视作一个协程，调用这个函数，会在运行时：

- 分配一块空间，存放协程整个生命周期内所需要的局部空间，存放用户定义的`promise`对象
- 返回一个用户定义类型的handle，这里是`task<return_type>`

原始的`handle`只由编译器提供两个功能：`resume`和`destroy`，前者在当前的栈中继续某个协程的执行，后者根据暂停位置，调用适当的析构函数，并在最后释放协程所占用的空间。

经过适当的包装后，我们可以给用户自定义的handle实现一个`operator co_await`，使其变为一个`awaitable`，这个函数返回一个`awaiter`供`co_await`使用，这样我们就可以在协程中调用其他协程了：`co_await task`

`awaiter`和`co_await`

一个完整的`awaiter`如下，其中的内容稍后解释

```
struct awaiter {
    bool await_ready() { return false; }
    std::coroutine_handle<> await_suspend(std::coroutine_handle<> h) {

        return std::noop_coroutine();
    }
    void await_resume() {}

};
```

编译器在协程的入口，`co_await`和`co_return`处创建暂停点。当执行`co_await <expr>`时，实际执行的内容是[1]：

- 计算表达式，获得一个`awaiter`
- 如果`awaiter.await_ready() == true`，则co_await执行完毕，否则继续

- 暂停当前协程
- 调用`awaiter.await_suspend(current_task_handle)`，这个函数是用户定义的，它会返回一个handle
- 恢复刚才返回handle的协程的执行，（如果返回的是`std::noop_coroutine()`）则返回到调用`handle.resume()`的地方（通常是调度器）
- ...
- 当原来的协程（无论何种原因）被继续执行时，调用`await_resume()`，并把这个函数的返回值作为整个`co_await`表达式的返回值
- 继续执行原来的协程

注意到编译器是不参与调用链的维护的，需要用户在调用新协程时，把它的调用者保存下来，等到新协程执行完成后，恢复其调用者的执行。

这里实现的风格为对称协程，除非用户主动`yield`，否则协程调用是不会引入调度器参与的。

动态内存分配

注意到co_await的第一步是计算表达式，这里通常是一个协程函数，因此会动态的创建一个协程，分配他的空间。用户态协程库的行为也是如此（`libcoro(gcc 10.3.0)`，`cppcoro(clang 10.0.0)`）：

```
coro::task<std::optional<uint64_t>> __internal_task3(int i) {
    ...

    co_return (x^(uint64_t)i)-y;
}

coro::task<std::optional<int>> __internal_task2_1(int i) {
    co_return co_await __internal_task3(i+1);
}

coro::task<std::optional<double>> __internal_task2_2(int k) {
    int x[100];
    ...

    uint64_t y = *co_await __internal_task3(k+1);

    co_return x[y%100] + y;


}

coro::task<std::optional<int>> __internal_task2_3(int i) {

    uint64_t y = *co_await __internal_task3(i+1);

    co_return (int)(y%100);

}

coro::task<int> __internal_task1(int i) {

    int x,y;
    if (i<10) {
        x = *co_await __internal_task2_1(i+1);
    } else {
```

```
        x = (int)*co_await __internal_task2_2(i+1);
    }

    y = *co_await __internal_task2_3(i+1);

}
```

```
alloc: 128
alloc: 96
__internal_task1: start
__internal_task1: before __internal_task2_1
alloc: 96
__internal_task2_1: before __internal_task3
alloc: 96
free: 0x5623452fe430
free: 0x5623452fe3c0
__internal_task1: before __internal_task2_3
alloc: 96
__internal_task2_3: before __internal_task3
alloc: 96
free: 0x5623452fe430
__internal_task2_3: y=-3430508659
free: 0x5623452fe3c0
__internal_task1: x=864458637, y=57
free: 0x5623452fe350
free: 0x5623452fe2c0
__internal_task1(2) output = 0
```

因为通常来说协程有某种传递性质，除了最上层外，只有协程才能够调用协程，不然会失去无栈协程的意义。因此一个协程任务在`yield`之前需要触发很多次动态内存分配。这样的动态内存分配对于一个无锁的现代`malloc`实现或许压力不大，但在我的简单内存分配器上对性能影响较大。

`Heap Allocation eLision Optimization`

> The call to `operator new` can be optimized out (even if custom allocator is used) if
>
> - The lifetime of the coroutine state is strictly nested within the lifetime of the caller, and
> - the size of coroutine frame is known at the call site
>
> in that case, coroutine state is embedded in the caller's stack frame (if the caller is an ordinary function) or coroutine state (if the caller is a coroutine)

据一篇标准库的文档[2]所述只需少数几个函数能被内联则可以触发这个优化，但实际上似乎并没有成功

优势

单个协程占用内存空间小，任务队列中可以有较多协程，在任务需要长时间阻塞时可以调度其他协程执行

```
    task<void> subtask(uint32 seed) {
        seed = complex_task(seed);

        co_await sleep_awaiter(timer::TICK_FREQ);

        seed = complex_task(seed);

        dummy += seed;

        subtask_done();

        co_return task_ok;
    }


    uint64 complex_task(uint64 seed){
        uint64 result = 0;
        for (uint64 i = 0; i < times; i++) {
            result += seed * 1103515245 + 12345;
        }
        return result;

    }
```

协程版本`100000`个计算任务，循环次数`100000`大约需要3s, 多线程版本大约40s。

```
[3528855] [ INFO] [C] [3] 144 us (diff 118 us) (create start)
[3529294] [ INFO] [C] [3] 428914 us (diff 428770 us) (create done)
[3529689] [ INFO] [C] [3] 3072524 us (diff 2643610 us) (done)
```

对于更实际的任务也有写相应的协程版本的测试，例如对1024个块进行同时读写。每个块读写100次。需要大约2秒钟

```
[3570121] [ INFO] [C] [1] 1145132 us (diff 366 us) (inc start)
[3570462] [ INFO] [C] [1] 3070690 us (diff 1925558 us) (done)
```

但是由于牵扯众多模块，没有写多线程版本的测试。

由于某些我实现的协程库的问题，协程的整体性能没有达到很理想的状态。

问题

- 过大的`task<return_type>`：在设计之初出于简便考虑把wait_queue的成员设计成了具有虚函数的类，于是task的虚函数表占用了较大空间。由于promise中也需要保存调用者的handle，使得promise的空间也变大了
- 错误处理引入的overhead:在设计之初希望在协程内部实现简洁的错误处理，然而可能因此引入了一些性能复杂度。例如返回值使用`std::optional`进行了包装，在返回时检测失败并向上寻找错

误处理的handler

- 协程任务队列使用共享链表，性能不佳
- 内存分配器比较简陋，不太适用于协程场景
- `gcc`对于协程空间的分配似乎比较保守，例如在如下情况，会为每个临时任务都分配8字节的空间，而实际上可以复用

```
x += co_await __internal_task3(i+2);
```

```
x += co_await __internal_task3(i+2);
x += co_await __internal_task3(i+3);
x += co_await __internal_task3(i+4);
x += co_await __internal_task3(i+5);
```

(后者比前者多占用24byte的空间，已去除每个新增暂停点的空间影响)

总结

- 引入协程后，在一个异步的实现下依然保持了同步阻塞的实现风格，从代码风格的角度没有引入太多压力。但是需要密切关注内存分配问题。
- c++语言复杂，容易不小心引入不必要的开销，例如我遇到的虚函数表，此外还有模板展开带来的缓存问题。

参考文献

1. https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await ↩

2. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html ↩