REPORT

周韧平 计11班 2021010699

Task 1

代码补全

serial_execute_all

```
void serial_execute_all()
    {
        is_parallel = false;
        g_pool = this;
        bool all_finished = 0;  //标记是否都结束
        while (!all_finished) { //只要存在没有结束的协程，就不断循环
            all_finished = 1;
            for (int i = 0; i < coroutines.size(); i++) {    //轮询所有状态
                if (!coroutines[i]->finished) { //发现没有结束的协程
                    all_finished = 0;
                    if(!coroutines[i]->ready){  //Task 2内容
                        coroutines[i]->ready = coroutines[i]->ready_func();
                    }
                    if (coroutines[i]->ready == true) {
                        context_id = i; //更新正在执行的协程id
                        coroutines[i]->resume();    //切入这个协程
                    }
                }
            }
        }
        for (auto context : coroutines) {//所有协程都执行完后删除所有context
            delete context;
        }
        coroutines.clear();
    }
```

"context.s"

```
.global coroutine_entry
coroutine_entry:
    movq %r13, %rdi
    callq *%r12

.global coroutine_switch
coroutine_switch:
    # TODO: Task 1
    # 保存 callee-saved 寄存器到 %rdi 指向的上下文
    # 保存的上下文中 rip 指向 ret 指令的地址（.coroutine_ret）
    leaq .coroutine_ret(%rip), %rax
    movq %rax, 120(%rdi)
    movq %rsp, 64(%rdi)
    movq %rbx, 72(%rdi)
    movq %rbp, 80(%rdi)
```

```
    movq %r12, 88(%rdi)
    movq %r13, 96(%rdi)
    movq %r14, 104(%rdi)
    movq %r15, 112(%rdi)

    # 从 %rsi 指向的上下文恢复 callee-saved 寄存器
    movq  64(%rsi), %rsp
    movq  72(%rsi), %rbx
    movq  80(%rsi), %rbp
    movq  88(%rsi), %r12
    movq  96(%rsi), %r13
    movq  104(%rsi), %r14
    movq  112(%rsi), %r15
    # 最后 jmpq 到上下文保存的 rip
    jmpq *120(%rsi)

.coroutine_ret:
    ret
```

resume与yield

```
void yield()
{
    if (!g_pool->is_parallel) {
        // 从 g_pool 中获取当前协程状态
        auto context = g_pool->coroutines[g_pool->context_id];
        // 调用 coroutine_switch 切换到 coroutine_pool 上下文
        coroutine_switch(context->callee_registers, context->caller_registers);
    }
}
virtual void resume() {
    coroutine_switch(caller_registers,callee_registers);
    // 调用 coroutine_switch
    // 在汇编中保存 callee-saved 寄存器，设置协程函数栈帧，然后将 rip 恢复到协程 yield 之后所需要执行的指令地址。
}
```

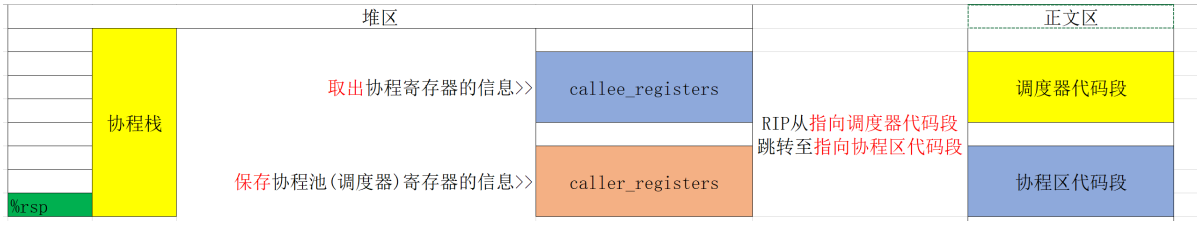
协程切入&切出时函数栈变化

协程通过callee_registers寄存器的赋值来实现"系统栈"和"协程栈"之间的切换

具体到每个协程栈来说，每次`resume`切入一个协程时，`coroutine_switch`会做三件事：

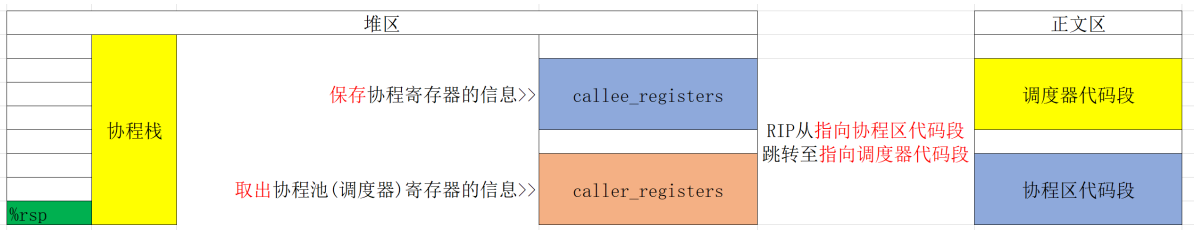
1. 将调度器运行时的寄存器状态（这里寄存器状态仅限被调用者保存的八个寄存器，下同）存储到`caller_registers`中

2. 从`callee_registers`取出上一次协程切出时保存的寄存器状态（第一次切入和非第一次切入`callee_registers`中RIP内容略有区别，上文对此有解释）赋给寄存器
3. 将原来指向协程池栈顶的%rsp指向协程栈栈顶（其实这包含在2中，因为%rsp对栈的定位很重要所以特意强调一下）



对于`yield`来说，`coroutine_switch`所做刚好相反：

1. 将协程运行时的寄存器状态保存到`callee_registers`中
2. 从`caller_registers`取出上一次切入协程前，也是调度器（协程池）执行时的寄存器状态赋给寄存器
3. 将原来指向协程栈顶的%rsp指回协程池栈栈顶，继续轮询



结合源代码，解释协程是如何开始执行的

"context.h"

```
basic_context(uint64_t stack_size)
      : finished(false), ready(true), stack_size(stack_size) {
    stack = new uint64_t[stack_size];
    // 对齐到 16 字节边界
    uint64_t rsp = (uint64_t)&stack[stack_size - 1];  //在堆区开一个模拟栈的数组
    rsp = rsp - (rsp & 0xF);  //对齐十六字节边界
    void coroutine_main(struct basic_context * context);
    callee_registers[(int)Registers::RSP] = rsp;
    // 协程入口是 coroutine_entry
    callee_registers[(int)Registers::RIP] = (uint64_t)coroutine_entry;
    // 设置 r12 寄存器为 coroutine_main 的地址
    callee_registers[(int)Registers::R12] = (uint64_t)coroutine_main;
    // 设置 r13 寄存器，用于 coroutine_main 的参数
    callee_registers[(int)Registers::R13] = (uint64_t)this;
  }

```

代码解释：程序运行一开始会调用`basic_context`的构造函数，构造函数中首先为这个context在堆区开了一个数组来模拟栈帧，同时将`callee_registers`数组的部分位置设定好初值，几个初值的作用如下：

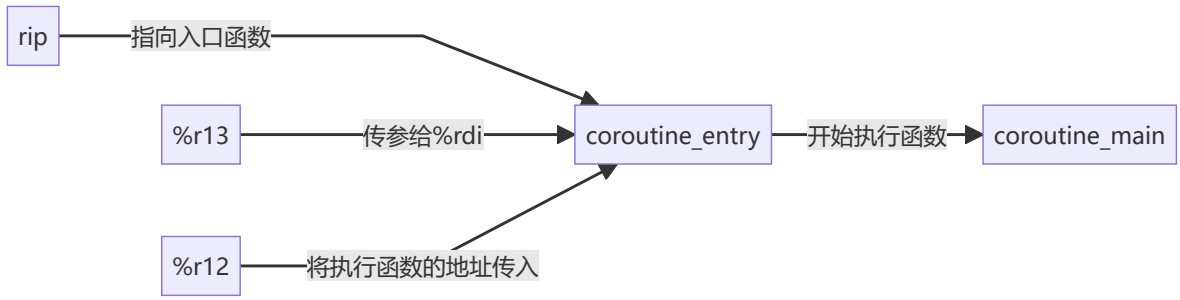
RSP：保存该协程的栈帧地址

R13：保存此context指向自己的指针

RIP：保存协程入口`coroutine_entry`函数地址，在这个函数里，会执行一步参数赋值操作("context.s"第三行)，然后再去执行`coroutine_main`

R12：保存`coroutine_main`的地址，也就是`run()`函数所在的位置

第一次调用`resume()`函数（同样也是第一次执行此`context`的`coroutine_switch`函数），此时`callee_registers`中的RIP指向的是`coroutine_entry`函数地址，于是会首先跳到`coroutine_entry`的中，将%r13(此时存的是指向这个context的指针)赋值给%rdi（相当于给`coroutine_main`函数的参数赋值），再跳到%r12所保存的`coroutine_main`的函数地址，执行这个协程的`run`函数的代码。



此后再执行`coroutine_switch`函数时，RIP都是重定向到第二个参数所保存的RIP值，对于`resume`来说，RIP从调度器代码段跳转到对应`context`的代码段，而对于`yield`来说则相反，RIP从`context`代码段跳转回调度器代码段。

coroutine_main:

```
void coroutine_main(struct basic_context *context) {     //第一次resume时进入
  context->run();   //执行该协程（对本体来说时show函数）,如果协程未执行结束，则会在函数内yield
  context->finished = true; //如果run函数顺利执行结束，则该协程已执行完成，此时修改执行标记
  coroutine_switch(context->callee_registers, context->caller_registers);//最后跳出协程，相当于最后一次yield
}
```

coroutine_main内包含的其实就是整个函数从开始执行到最终结束的过程。

Task 2

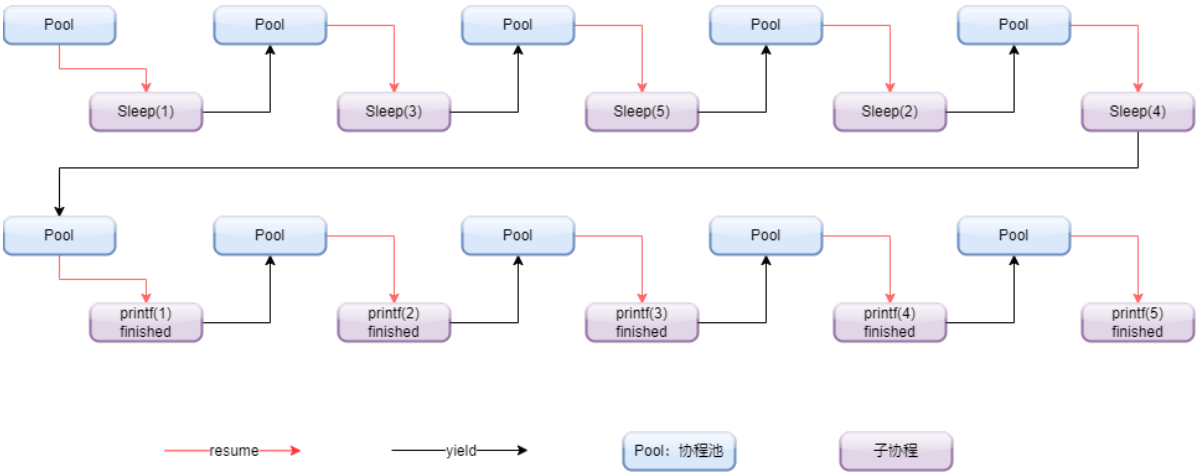
代码补全

```
void serial_execute_all()
    {
        is_parallel = false;
        g_pool = this;
        bool all_finished = 0;  //标记是否都结束
        while (!all_finished) {
            all_finished = 1;
            for (int i = 0; i < coroutines.size(); i++) {    //轮询所有状态
                if (!coroutines[i]->finished) { //轮询到一个未完成的协程
                    all_finished = 0;
                    if(!coroutines[i]->ready){  //如果该协程还没有“睡醒”，则需要判断一下它是否需要“睡醒”，即执行ready_func
```

```
                        coroutines[i]->ready = coroutines[i]->ready_func();
                    }
                    if (coroutines[i]->ready == true) { //如果协程“睡醒”，则执行它
                        context_id = i; //更新正在执行的协程id
                        coroutines[i]->resume();    //切入这个协程
                    }
                }
            }
        }
    }
```

```
void sleep(uint64_t ms)
{
    if (g_pool->is_parallel) {  //并行
        auto cur = get_time();
        while (
            std::chrono::duration_cast<std::chrono::milliseconds>(get_time() - cur)
                .count()
            < ms)
            ;
    } else {    //顺序执行
        // 从 g_pool 中获取当前协程状态
        auto context = g_pool->coroutines[g_pool->context_id];
        // 获取当前时间，更新 ready_func
        auto cur = get_time();
        context->ready = false;
        // ready_func：检查当前时间，如果已经超时，则返回 true
        context->ready_func = [cur,ms](){
            return (
            std::chrono::duration_cast<std::chrono::milliseconds>(get_time() - cur)
                .count()
            >= ms);
        };
        // 调用 coroutine_switch 切换到 coroutine_pool 上下文，此处通过yield函数实现
        yield();
    }
}
```

按照时间线，绘制出 `sleep_sort` 中不同协程的运行情况



目前的协程库实现方式是轮询 `ready_func` 是否等于 `true`，设计一下，能否有更加高效的方法

可以尝试为每个协程设置优先级，对数值较小者，设置其协程优先级更高，而当此协程执行完后，通过适当的设置pass的值, 将其优先级设为最低或，这样，每次只要通过线性查找优先级最高者选择执行，即可起到和 `sleep` 函数相同的效果。

在此思想上进一步优化，可以通过维护一个类似于优先级队列的数据结构，来减小每次对线性扫描查找到优先级最高者而产生的时间开销，这样可以将每次查找的时间复杂度优化到均摊$O(logn)$。

Task 3

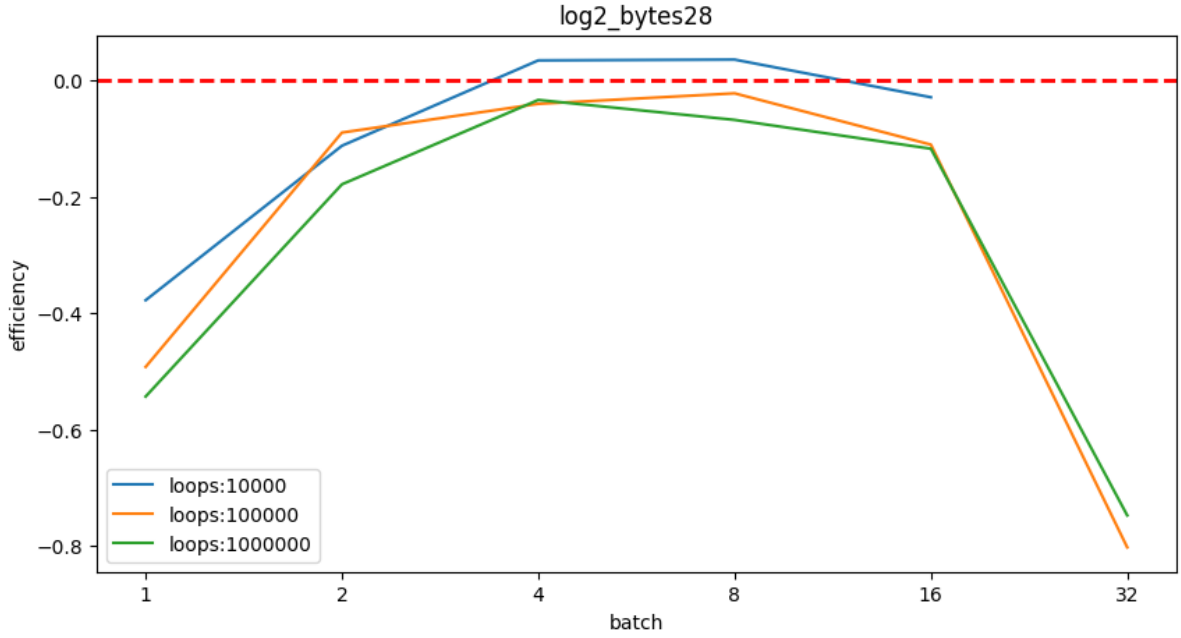
代码补全

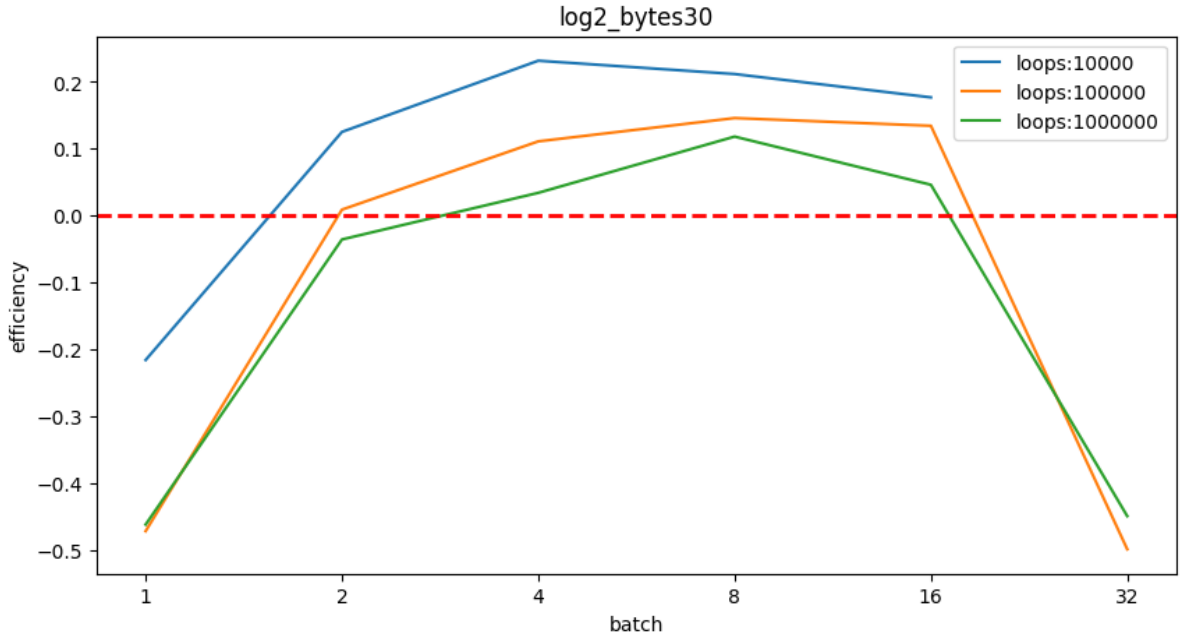
```
while ((size / 2) > 0) {
  size_t half = size / 2;
  size_t probe = low + half;
  // TODO: Task 3
  // 使用 __builtin_prefetch 预取容易产生缓存缺失的内存
  __builtin_prefetch(&table[probe]);
  // 并调用 yield
  yield();
  uint32_t v = table[probe];
  if (v <= value) {
   low = probe;
  }
  size -= half;
 }
```

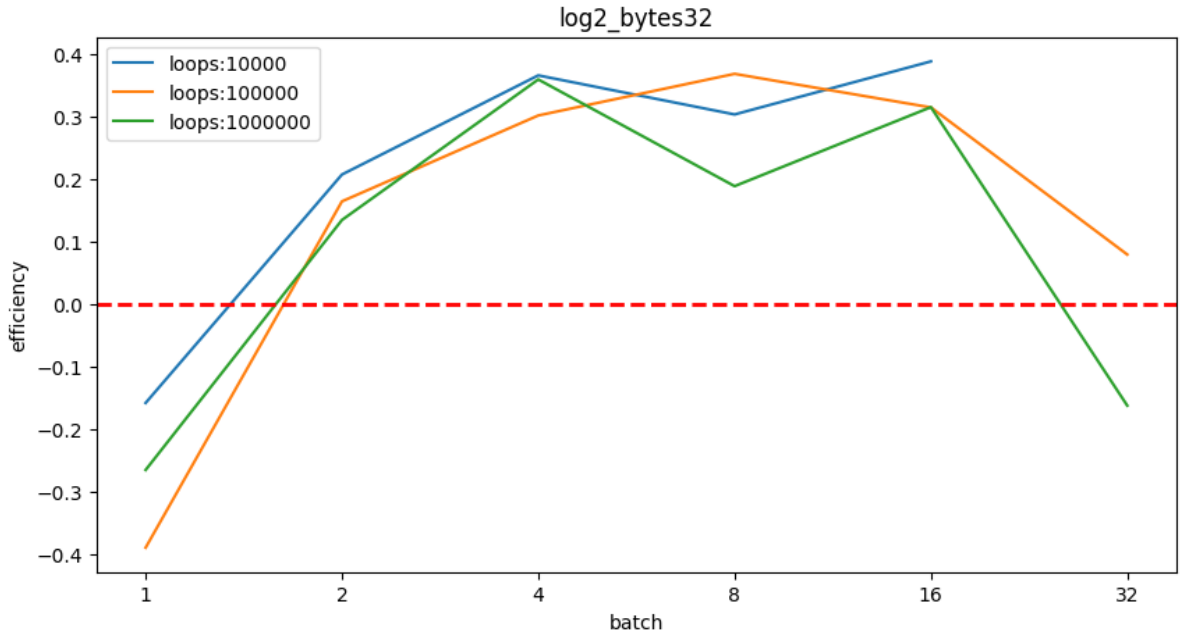
汇报性能的提升效果

此部分实验我借助了宋曦轩同学提供的测试脚本（脚本保存在 `coroutinelab/coroutinelab/experiments.py`），测试loops在28，30，32下的优化效果（每种参数设定做10次实验取平均值），实验数据保存在 `coroutinelab/coroutinelab/experiment_tiny_results.json` 中。

根据实验数据绘制实验结果如下，其中纵坐标efficiency表示使用协程优化效率，计算公式为 $efficiency = \frac{coroutine\ access - naive\ access}{naive\ access}$







数据可以观察得出以下结论：

1. 固定Log2_bytes，**协程开的个数batch对二分查找的优化效率呈先增后减的趋势，极值点出现在batch为4~16**，猜测batch较大时优化效率低的原因可能是创建和切换协程的时间成本要高于提前缓存节省的时间消耗，batch较小时优化效率低则可能是协程数量少优化效果不明显，而创建协程和切换本身又会造成时间消耗有关
2. 对比不同的Log2_bytes，**Log2_bytes越大整体优化效果越好**，比如在Log2_bytes为28时几乎全部为负优化，而Log2_bytes为32时除了batch=1和barch=32时，二分查找的效率均得到了一定的提升，这可能是因为在数组较小时需要预取的缓存较少，导致优化不明显。

除了上述结果，在获取实验数据时，注意到**优化性能波动比较大**，在一些特殊情况下，协程负优化相当明显，**算法对输入较为敏感。**

总结

感谢助教提供的保姆级教程，实验的代码部分完成的相当顺利，但是花在REPORT上的时间则远远超出了自己的预期，其中的一个原因可能是REPORT完成断断续续，每次都得重新研究一遍才能回想起代码的逻辑来；另一个原因是花了太多的时间在各种作图上，通过这次作业找到了一些不错的绘图软件，希望之后的Lab不会在这些事上花太多的时间。

感谢宋曦轩同学提供的轮子，帮我省去了不少花在pyhton上的时间

qwq下次一定要争取一口气写完REPORT（Flag必倒系列）